(4)

LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TR-434

A Programming Language Supporting First-Class Parallel Environments

Suresh Jagannathan

January 1989

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

4

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR-434	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-84-K-0099
6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO | WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
A Programming Language Supporting First-Class Parallel Environments

12. PERSONAL AUTHOR(S)
Jagannathan, Suresh

13a. TYPE OF REPORT Technical	13b. TIME COVERED FROM ______ TO ______	14. DATE OF REPORT (Year, Month, Day) 1989 January	15. PAGE COUNT 207

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Language Design, Modularity, Namespace Management, First-Class Environments, Non-strictness, Non-Determinism, Programming Environments, Type Interface, Dataflow Languages

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Namespace management is fundamental (in a practical sense) to the design of any programming language: how are naming environments built, and how are they used? Modern programming languages come equipped with a variety of mechanisms to create and manipulate naming environments. These mechanisms fall into two broad categories: program structures and data structures.

Program structures and data structures are treated differently in modern programming languages. Program structures are not considered to be data structures: they cannot be examined, nor can they be used as components of other data structures. Nor are data structures considered to be programs: they do not specify a scope, nor can they contain expressions as primitive components.

This thesis presents a new programming model called the symmetric model in which the (cont.)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b. TELEPHONE (Include Area Code) (617) 253-5894	22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

U.S. Government Printing Office: 1986—507-047

89 5 12 003

19.

representation of programs is identical to the representation of data: to specify a computation, one defines a data structure. This data structure possesses the semantics of a first-class naming environment-it defines a scope and can be used to affect the evaluation environment of other expressions.

We present a new programming language called Symmetric Lisp based on the symmetric model. Program structures in Symmetric Lisp are considered non-strict: a program's components may be examined even as its other elements continue to evaluate. The first part ot the thesis investigates the interaction of non-strictness with first-class naming environments. The second part of the thesis discuss the compilation and implementation of Symmetric Lisp. We present an extended type-inference system for first-class environments that can be used to infer the proper evaluation environment of identifiers found within the scope of environment-yielding expressions. We also present a translation of Symmetric Lisp into a high-level dataflow language.

A Programming Language Supporting First-Class Parallel Environments

Suresh Jagannathan

MIT / LCS / TR-434
January, 1989

This report describes research done at the Laboratory of Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-84-K-0099.

This report is also published as the author's doctoral dissertation.

A Programming Language Supporting First-Class Parallel Environments

Suresh Jagannathan

Technical Report MIT / LCS / TR-434
January, 1989

MIT Laboratory for Computer Science
545 Technology Square
Cambridge MA 02139

Abstract

Namespace management is fundamental (in a practical sense) to the design of any programming language: how are naming environments built, and how are they used? Modern programming languages come equipped with a variety of mechanisms to create and manipulate naming environments. These mechanisms fall into two broad categories: *program structures* and *data structures.*

Program structures and data structures are treated differently in modern programming languages. Program structures are not considered to be data structures: they cannot be examined, nor can they be used as components of other data structures. Nor are data structures considered to be programs: they do not specify a scope, nor can they contain expressions as primitive components.

This thesis presents a new programming model called the *symmetric model* in which the representation of programs is identical to the representation of data: to specify a computation, one defines a data structure. This data structure possesses the semantics of a *first-class naming environment* – it defines a scope and can be used to affect the evaluation environment of other expressions.

We present a new programming language called Symmetric Lisp based on the symmetric model. Program structures in Symmetric Lisp are considered *non-strict*: a program's components may be examined even as its other elements continue to evaluate. The first part of the thesis investigates the interaction of non-strictness with first-class naming environments. The second part of the thesis discuss the compilation and implementation of Symmetric Lisp. We present an extended type-inference system for first-class environments that can be used to infer the proper evaluation environment of identifiers found within the scope of environment-yielding expressions. We also present a translation of Symmetric Lisp into a high-level dataflow language.

Key Words and Phrases: Language Design, Modularity, Namespace Management, First-Class Environments, Non-strictness, Non-Determinism, Programming Environments, Type Inference, Dataflow Languages.

Acknowledgements

He who posseses faith attains knowledge
Devoted to that (knowledge), restraining his senses
Having attained knowledge, he goes speedily
To Supreme Peace

Bhagavad-Gita, IV-39

The ideas for this thesis grew out of a long series of discussions I had with Prof. David Gelernter during the summer of 1986 at Bell Laboratories on programming languages, parallelism and uniformity. I am grateful to David for the many important ideas he has given me since then and for the confidence he has shown in my work.

When I returned to MIT in the Fall of 1986, my ideas were ill-formed and nebulous. Transforming my vague intuition into a well-defined dissertation would not have been possible had it not been for the efforts of my thesis advisor, Prof. R.S. Nikhil. During the two and one-half years that I have worked on this thesis, Nikhil has taught me much about programming languages, the meaning of scientific research, and the importance of precision and accuracy in exposition. I am indebted to him for the interest he has taken in this work, and for the countless improvements he has made to the quality of this document.

My two thesis readers, Prof. Robert Halstead and Prof. Gerald Sussman took an active interest in my research and encouraged me to pursue novel and worthwhile ideas I would not have otherwise thought of. I am especially thankful to Bert for his meticulous reading of the thesis and for his numerous suggestions on how to improve it.

In addition to my readers, it is a pleasure to acknowledge the many insightful comments Mark Day has given throughout the course of this research.

I was very fortunate to undertake this work while a member of the Computation Structures Group whose members collectively contribute to form an exceptionally stimulating and friendly environment. In particular, I am thankful to Ken Traub for the extensive TEX macro facilities he has implemented which were used to format this thesis, and to Shail Aditya Gupta for the many conversations we have had on this work.

Besides the fine technical environment in which this work was undertaken, I have also had the good fortune of having roomates who always inspired sparkling conversation and never faltered in their wit. My thanks to both C.S. Venkatakrishnan and Sumanth Kaushik for their special contributions to my social well-being.

Above all, the research documented in this thesis could not have been conceived, let alone written, had it not been for the unwavering support and encouragement given to me by parents, my brother Aravind, and Hema; it is to them that this thesis is dedicated.

Contents

Chapter 1

Introduction

A practical concern in the design of any programming language is the issue of namespace management[1]: what are the mechanisms by which naming environments are built, and what constraints need to be imposed to use them sensibly? Modern day programming languages come equipped with a variety of mechanisms to create and manipulate naming environments. These mechanisms fall into two broad categories: *program structures* and *data structures.*

A program structure specifies a *computation*. Modules, begin-end blocks, packages, and classes are examples of program structures. The expressions or statements evaluated within a computation communicate their results through a *naming environment* that is built and maintained by the program's implementation. A naming environment is typically represented as a set of *bindings* or name-value pairs that associate a name with the value it denotes in the program; thus, the names found within an environment are the names declared within the program. An environment is used as a *namespace* or *scope* within which evaluating expressions execute.

A data structure specifies a value or a collection of values. Records, arrays, streams and lists are examples of data structures. Because data structures specify values, they can be examined by expressions, passed as arguments to functions, or incorporated as components of other data structures. Some data structures can also define names; a record, for example, defines a

[1] "Practical" as opposed to "fundamental": it is not theoretically necessary for a programming language to support a notion of a naming environment. A pure combinator-based language such as FP[12] is a case in point. With a suitable number of primitive functions and functional forms, it is possible to write every computable function in a name-free language like FP. In a practical sense, however, the inability to write history-sensitive programs in such languages makes them strictly less expressive than languages that do support naming environments. It is for this reason that extensions to FP (such as FFP and AST) have been proposed that allow the programmer to write new combining forms and history-sensitive programs; these extensions do support (albeit implicitly) a notion of a naming environment.

collection of heterogeneous fields in which the value of a field is accessed through a particular name.

Program structures and data structures are treated differently in modern programming languages. Naming environments are not considered as data structures: they cannot be examined, nor can they be used as components of other data objects. Data structures are not viewed as programs: data structures do not define a scope nor can they contain expressions as their primitive components. In particular, the names defined in a naming environment are treated differently from those defined in a data structure such as a record – while a name binding found in a local naming block can be used (implicitly) to affect the evaluation of expressions, a binding found in a record cannot. These distinctions lead to a programming model in which "program" is viewed as orthogonal to "data"; the implementation of a program structure in terms of a naming environment is semantically unrelated to the way data structures are used and represented.

This thesis presents a new programming model that is based on a very different notion of what programs and data structures ought to be. In the *symmetric* model, the representation of programs is identical to the representation of data: to specify a computation, one defines a data structure. This structure can be examined and can be used as a component of other structures, yet it has the same semantics as a naming environment – it defines a scope and can be used to affect the evaluation of other expressions. A data structure in this model is a program in which all component elements are values. In this model, there is only one kind of name and only one way to associate a name with a value. Data and program have the same syntactic structure and bear the same semantic import.

In order to investigate and judge the ramifications of such a model on program building in general, we also develop a new programming language that provides direct support for the symmetric model. Although one can conceive of many different interpreters for a symmetric language, each implementing a different evaluation strategy, the operational semantics developed in this thesis views program structures as *non-strict*: a program's components may be examined (since a program is a data object) even as its other elements continue to evaluate. Under such a semantics, whenever two elements of a symmetric program are "spatially distinct" (occur as separate components of the same data structure), they may evaluate simultaneously. Parallelism of this sort combines with first-class naming environments to form an interesting synergy, and the investigation of their interaction forms the bulk of the thesis.

The central proposition of this thesis is that a symmetric programming model obviates the need for the multitude of special-purpose program-building and data-constructing forms found in languages based on other programming models; it is argued that the behaviour of these forms can be subsumed within a single, simply described structure that possesses the salient characteristics of both program and data.

1.1 Intuition

A program in a symmetric model is a *map*. To write a program, one draws a map of the computation and hands this map to an interpreter for evaluation. A map is a collection of heterogeneous *regions* that are connected in a well-defined way. Each region may have a name; the contents of the region may contain either a value or a reducible expression. Because we view a symmetric language program as a parallel program, the role of the symmetric language interpreter is to develop the map (metaphorically) in the same way that a photographic negative is developed, by operating on the entire surface simultaneously and in a uniform way. The object returned by the interpreter is the same map with the contents of each region filled in with the value denoted by the expression previously occupying the region.

A map containing reducible expressions is a "program-map"; a fully-developed map is a "data structure".

Each region of a program-map may have a single name; but regions (even when unnamed) always occupy unique positions. All names found within a map must be unique. Figure 1 indicates the structure of a general map. The figure defines a map with four regions in which the first region is named $N1$, the third region, named $N3$, itself consists of a map containing two regions named $N3 - 1$ and $N3 - 2$ resp. The second region is unnamed.

Each region of the program-map contains either a primitive expression (*e.g.*, a conditional or arithmetic expression or function definition) or another, nested program-map. Every name referred to within a primitive expression is either wired into the interpreter (like the names "if" or "plus", say), or is the name of some other region. The named region may be in a map enclosing the expression which references it or explicitly specified as part of the expression's evaluation environment.

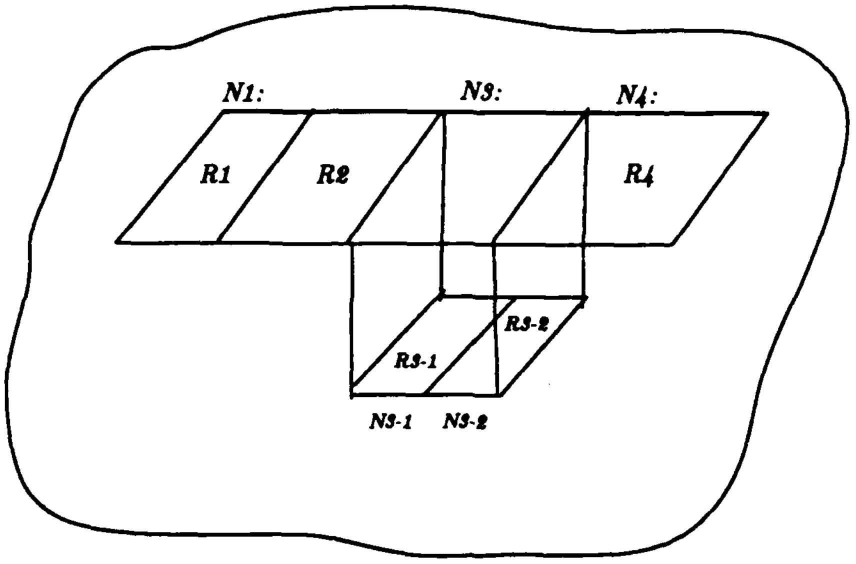

Figure 1 : Abstract Structure of a Symmetric Language Map

Expressions may refer to regions by name. If an expression found in region R refers to the name N, then the interpreter resolves the reference by first searching for a region labelled N in the map in which R resides. If such a region exists, its contents is returned and the search terminates. If no such region is found in the current map, the search proceeds upward to the map (call it M) enclosing the map in which R is found; if N is not defined in this map, the search continues upward to the map enclosing the map of which M is a part; the search process continues in this fashion, until a region with label N is encountered whereupon the search terminates and the contents of its region is returned.

When an expression refers to the name N, then N refers to "the contents of the region labelled N *after* it has has evaluated to a value". This means that, in developing a map, the evaluation of a region blocks only if it requires the contents of another region still under evaluation. The same rules apply recursively to maps nested within regions, so that in general, an expression within a region may refer not only to names in its own sub-map but to names of regions defined in enclosing super-maps.

The data structure generated from a program-map has the same structure as the program. But every expression in the program-map has been replaced by its value, as determined by the evaluation rules discussed later. Every data structure map is a program that doesn't change when it is evaluated, a program that is in "normal form".

Note that names are treated the same way regardless of whether they are found in a data structure or a program. Names defined in program structures can be evaluated to yield the value to which they are bound; names defined in data structures can be used to affect the evaluation of other expressions because the data structure map itself defines a scope.

How does this model relate to an archetypical programming model such as the lambda calculus? In the lambda calculus, one views programs and data structures as functions that map from some set of input values to some set of output values. In the symmetric model, one views programs and data structures as maps that, unlike functions, have a well-defined spatial interpretation. In the symmetric model, there is no distinction between a program structure and a data structure: every program is a data structure with a certain "shape" and physical manifestation. In the lambda calculus, every data structure is a program that represents some computable function.

1.2 Why Symmetry?

What is "symmetric" about the symmetric programming model? Consider the semantics of conventional data structures with respect to a space-time axis. Elements of a data structure occupy distinct points along the space axis (hence, we can refer to the i^{th} element of vector V, or to the third field in record R). While the n elements of a data structure occupy n distinct points along the space line, all n points usually occupy the same point along the time axis in the sense that they all share the same extent. The elements of a record, for example, are usually created together and their lifetime is usually fixed to be the lifetime of the record itself. Now consider a basic program structure like a (sequential) compound block or statement sequence. The elements of a block occupy distinct points along the time axis (hence, we can make assertions of the form, "expression i evaluates after expression $i+1$"). It is usually not possible to transpose the order in which elements of a block are executed[2]. On the other hand, all elements of a compound block occupy the same point along the space axis: once an expression has finished evaluating, the space it occupies can be reclaimed for use by the expression following it.

Hence, we see that data structures and program structures can be related to another in a

[2]This assertion is predicated on the assumption that expressions can side-effect the data they reference.

natural way: they are symmetric transposes with respect to a space-time axis. In looking at structures relative to their behaviour along this axis, we see that the symmetric language map is the symbiosis of the compound block and the multi-field record: it is a shared-lifetime, separate-space data structure.

1.3 Implications

What practical benefits does the symmetric programming model offer that other models do not? Uniformity of program and data is the key difference: using the symmetric programming model, one can devise useful programming paradigms that, in a very practical sense, depend on a uniform treatment of program structures and data structures. We contend that the realization of programs based on these paradigms would, in fact, be hindered were such uniformity not an integral part of the programming model. Uniformity manifests itself in a number of different ways:

Uniformity of Diverse Program Structures:

Because maps are the basic program structure in this model and simultaneously also the basic data structure, programs can be written as maps and then examined and returned. One can iterate over map versions of arbitrary records, blocks, modules or packages in the same way. That a map represents a particular spatial arrangement is of importance beyond the fact that it can be used to represent a conventional data structure. The activation structure of a symmetric language function is represented as a map, as is the structure in which its actual and formal parameters are stored. It is *not* the case that one structure (the map) happens to be useful in two unrelated domains (program and data structures); once the the artificial boundary separating the two is torn down, many novel results follow. For example: if a program is written as a linear map named *Prog*, it can be treated as a vector; once *Prog* has completed, one can iterate over the entire structure by examining *Prog*[i] for all i. The arrangement of definitions and expressions in a program is not a random attribute of some text file, but instead represents a programmer decision with implications at runtime[3].

[3]Of course, expressivity of this sort also comes with some costs. It may not be possible, for example, to build an optimizing compiler that reorders or combines elements in a map to improve performance; we discuss the implementation problems that a symmetric language faces in Chapter 6.

To illustrate this point, consider a program that defines a record object R. Suppose that several of R's fields are to be function-valued. Suppose, further, that we require many different instances of R and, moreover, that each such instance is to be held in a specified library. We also wish to add a new field to R that does not contain a passive data object, but an executable statement, a loop, for example, that constantly monitors the status of R's other fields.

In most other languages, implementing such a structure would have required using a number of different (and, often unrelated) module-structures: records to represent R's skeleton, packages[3] to implement libraries, classes[23] to implement templates, and processes to implement R's active components. A symmetric model, on the other hand, treats all these modularity devices in a uniform and consistent way. The symmetric language programmer can preserve his idea of modularity even as he expands the functionality of the basic module unit; he need not switch from one superficially different module construct to another (and another, and so on).

Uniformity of Language-Level and System-Level Structures:

Uniformity of program and data has non-trivial implications for program building when considered in the context of a semantics that defines a non-strict evaluation strategy. For example, it is possible to express programs that turn into conventional data structures when they are complete: a program called *AtimesB*, for example, might be a rectangular map whose i, jth element is an invocation of the function *InnerProduct* on the i^{th} row of A and j^{th} column of B; more complex "active data structure" programs might have certain elements that block pending the completion of other elements. It is thus possible to encompass the kind of programming style that eager, non-strict, parallel programming languages[26, 40, 53] support.

As a concrete example, consider the following situation: the user wants to run many test cases of a program Q concurrently; he wants to analyze the results of each using an analysis program *analyze*; whenever an analysis turns up a "best result so far", he wants the results entered in a *best-results* directory and a message printed to the terminal screen. Moreover, if Q and *analyze* are compute-intensive, it becomes desirable to have Q's and *analyze*'s internal structure be as parallel as possible; we would like to constrain parallelism only when the logical dependency constraints of the program would be violated.

Some of the requirements imposed by this example can be handled easily enough with the help of a conventional operating system. Q and *analyze* can be implemented as separate program units that are assembled dynamically in the style of piped-together routines in a Unix shell script. The *best-results* directory can also be created and maintained under the aegis of the operating system. The internal parallelism required to implement Q and *analyze* would, of course, have to be supported by the source language.

In most systems, however, there is a sharp distinction between the facilities provided by the language proper and that provided by the operating system[4]. Languages rarely allow loosely-knit computation ensembles (like Q and *analyze*) to coalesce and interact in the same way that closely-knit program units are allowed to. In a symmetric language, on the other hand, distinctions between the operating system domain and the programming language domain are blurred. First-class naming environments can be made to serve as file-systems; independently conceived program units can be dynamically assembled by enclosing them within a single map; the manner in which two elements in a single map relate to one another is similar to the way in which two program units found within a large directory relate.

Uniformity of Data and Process:

Thinking about programs in terms of maps also enables the programmer to have a unified framework for dealing not only with programs as objects but also with evaluating programs as processes. The same structure that is used to store the fields of a record or the elements of an array can also be used for the components of a computation in progress, as we illustrated above, simply because the top-level source program itself is represented as such a map, and the structure of this map is preserved during evaluation. An ongoing computation is a data object with all the power of maps generally.

Because a program is simply a data object, one can nest programs within other programs in the same way that one can nest data structures within other data structures. The internal structure of a nested program can be examined even if it is still in mid-evaluation. Viewing programs as *transparent* data objects encourages an interesting program methodology not supported by other programming models. For example, one can write a program essentially unencumbered by calls to i/o routines. Programmers are free to drape this program with routines that monitor its evaluation (since it is, after all, just a data object) and format and display results as they

[4]Monolingual workstations like Lisp machines[39] are an exception. But, even in such systems, the object specified by the file-system has no truly meaningful interpretation in the context of conventional lisp data structures.

see fit. Many different display routines can be written for the same program; the original core is left untouched.

As an example of where such a programming paradigm would be useful, consider an expert system that continuously monitors and filters information gathered from a large number of independent data streams. Internally, such a program might be organized as a collection of concurrently executing processes that communicate with one another via different data streams. Externally, we would like to treat the program as a large dynamic record whose individual fields represent independent decision nodes. We should like to be able to monitor the state of a given node or retrieve the contents of a particular data stream dynamically and from the outside without having to alter the structure of the program.

A symmetric language allows expert systems of this sort to be built using maps. Here again, the uniform structure of program structures and data structures enables the programmer to have different views of the same structure (internally, as a collection of processes; externally, as a large record-like object) as suits his convenience.

1.4 Symmetric Lisp

Symmetric Lisp is a programming language that directly supports the programming model described above. A detailed informal description is given in Chapter 2 and a formal semantics is provided in Chapter 3; what follows below is a summary of its fundamental features:

1. The only program and data constructor in the language is a map. A map defines a program structure because (a) it is an object that specifies a computation and (b) because the names it defines are used in the evaluation of other expressions. A map defines a data structure because (a) it allows its components to be selected by name as well as position and (b) because it may be treated as an aggregate structure. Map structures are always shared among expressions, never copied.

   Map evaluation is *non-strict*: the components of a map may be selected even as other components in the same map continue to evaluate. All expressions within a map evaluate concurrently subject only to basic data dependencies. It is possible to serialize the behaviour of map expressions by using a variant of the basic map constructor[5].

2. The language supports the definition and use of first-class *naming environments*. Because maps may be nested inside other maps and may be bound to names, environments may be

[5]*serialize* is not used here in the traditional database sense[30]; it is used to simply mean that within a "serialized" map, a left-to-right evaluation semantics of expressions is obeyed. The regions found in a regular map, in contrast, are not evaluated in any particular order.

yielded and treated as values. Symmetric Lisp allows the programmer to specify explicitly the naming environment within which an expression should evaluate; the meaning of any names not defined by the expression are retrieved from the user-specified environment. The meaning of names found in an expression that is evaluated in the context of a specified map is the value to which these names are bound in the containing their definition.

One can layer maps; the effect of such an operation is to have bindings of names found in one map supersede those defined in the other. One can also project names onto an environment; projecting a name n onto an environment M results in a new environment that contains n's binding in M.

3. There are operations to abstract over environments and expressions to create functions. Symmetric Lisp supports first-class, higher-order lexically-scoped functions. Environment and function abstraction are primitive operations in the language, but the application rules for abstracted expressions are defined in terms of map instantiation and selection.

4. Regions in a map may be left empty. An empty region or *hole* may be subsequently filled in by another map expression. Any expression that attempts to access a hole suspends, as per the normal evaluation rule, until a value is produced for that hole.

5. Regions are assignable. Assigning to a region only changes the contents of the region and not the name bound to that region. All assignment operations are atomic; simultaneous assignments to the same variable take place in arbitrary order.

1.5 Background

The symmetric programming model is the outgrowth of an investigation into the underlying similarities that exist among many superfically diverse programming methodologies. In this section, the symmetric model (and Symmetric Lisp) is compared with other related models and languages. The discussion of related models falls into two broad categories: (1) programming models that address the issue of program/data uniformity and (2) programming models that are based on a parallel evaluation semantics.

1.5.1 Programming Models

1.5.1.1 Programs as Objects

The object-oriented programming model[4, 38] is characterized by its treatment of both program and data as *objects* that retain local state information. Programs are conceived as a collection of objects that communicate by message-passing. In many ways, the symmetric model is the logical inverse of the object-oriented model. Every data structure in an object-based model is treated as a program whose internal state is only accessible via the methods defined for the object; in contrast, a symmetric language program is a data structure whose elements may be accessed directly without requiring any intervening method invocation.

Uniformity of program and data in the object-oriented model is achieved by transforming every data object into a program; uniformity in the symmetric model is achieved by transforming every program structure into a data object. Modularity in an object-oriented program is achieved by encapsulating information inside objects and using inheritance relations among objects to share common information; modularity in a symmetric language program is achieved by collecting related information into maps and allowing larger maps to be synthesized out of smaller ones by permitting arbitrary nesting of map structures.

The methodology imposed by object-based languages is based on encapsulation; decisions about what to do with messages and instance variables are determined statically by the class definition of the object. The methodology encouraged by a symmetric language is based on program transparency; decisions about what to do with data and program can be determined dynamically by expressions found outside of the object. As a program evolves, one can treat the objects it defines in different ways; the same program may have different routines operating on it in different ways, *e.g.*, displaying its internal state in diverse formats, etc.

Which approach is better? There are advantages as well as disadvantages to both programming methodologies. Several of the applications considered in this thesis, however, (*e.g.*, monolingual programming environments or dynamically evolving expert systems), crucially depend on program transparency; encapsulation of the kind encouraged by conventional object-based systems would be a hindrance in the development of such programs. On the other hand, it cannot be denied that encapsulation and abstraction serve a useful role in the construction and maintenance of large programs; we discuss how one might implement a simple form of data abstraction in the context of Symmetric Lisp later in the thesis.

1.5.1.2 Procedures as Data – Comparison with Lisp

Classical Lisp[48, 54] has been altered and extended in many ways. Scheme and its dialects[1, 31] are alterations designed to support higher-order functions and continuations. Common Lisp[57], among other things, provides modularity aids through the package structure. CommonLoops[14] and Flavors[39] are extensions for object-oriented programming. 3-Lisp[56] and Brown[67] are dialects that focus on user control of the interpreter's internal state.

Symmetric Lisp addresses many of these issues, albeit in a significantly different way; it departs from all these Lisps in its insistence on a uniform naming discipline for both programs and data. Lisp maintains a distinction between the names defined in program structures such as procedures or environments. and names found in data structures such as lists or alists. A Lisp procedure or a Scheme environment defines a collection of *program-structure* names – names that can be used to affect the evaluation of other expressions, *i.e.*, names that can be evaluated to yield their binding values. The names found in a Lisp data structure such as an alist or property-list, on the other hand, cannot be evaluated. The name-value pairs found in an alist do not have the semantics normally associated with a naming environment – they cannot be used to affect the evaluation of other expressions (since they do not define a scope) nor can they be handed to the interpreter for evaluation. In Symmetric Lisp, there is only one kind of name and only one way to associate a name with an object. Every name, whether found in maps used as data or in maps used as programs, has a value.

Scheme and First-Class Environments: In MIT Scheme[1], users are allowed to build customized environment structures through the `make-environment` special-form. `Make-environment` constructs an environment object, evaluates a sequence of expressions within this environment, and returns the new environment as its result. The `define` special form installs a binding within the environment in which it is evaluated; users access bindings defined in a particular environment by passing the environment as the second argument to the primitive `eval` operator. Thus, given an environment, `E`, containing definitions `x` and `y`, the expression: `(eval '(x y) E)` evaluates the application `(x y)` in the context of `E`.

The `make-environment` construct makes it possible for Scheme programmers to define local namespaces that encapsulate a related collection of data and procedures within a structure similar to the map objects constructible in Symmetric Lisp. Note that because `make-environment` yields a value (an environment object) which can be bound to other names, one can nest environments to any depth. Since environments can be bound to names and are *bona-fide* objects in the language, they may be built into data structures or they may be passed as arguments to, and returned as results from, procedures.

Given that Scheme provides a mechanism for building first-class environments, one is led to ask how these environment structures are different from Symmetric Lisp maps. Outside of the obvious difference between Scheme's strict, sequential evaluation semantics and Symmetric Lisp's non-strict, parallel one, Symmetric Lisp's implementation of naming environments differs from Scheme's in one other important way: Symmetric Lisp provides *transparent* access to the environment structures built by map evaluation; Scheme does *not* provide transparent access to its user-defined environments. Transparency means that the Symmetric Lisp map has a well-defined notation in the language; the Scheme environment object does not. To access the elements of a Scheme environment one must either evaluate the name whose binding-value is sought using the `eval` function or coerce the environment structure into a list or alist object. A Symmetric Lisp environment defined by map evaluation requires no coercion in order to be examined.

3-Lisp, Brown, and Reflection: Smith has proposed a dialect of Scheme called 3-Lisp that allows the hidden store, environment and continuation structures found in a Lisp interpreter to be made visible for explicit manipulation and inspection within Lisp itself. The reflective metaphor suggests an infinite tower of interpreters: the lowest level representing the interpreter that interprets user programs, the level above it representing the interpreter that has access to the structures used by the level 0 interpreter, and so on *ad infinitum*. Friedman and Wand have proposed a less metaphysical version of Smith's model that does not depend on a tower of interpreters; their language, Brown, requires only two levels of interpretation – the base interpreter and a *reified* interpreter that has access to the store, environment and continuation of the lower level.

It may be argued that any language is potentially "reflective": a Pascal debugger written in Pascal, for example, acts in much the same way as a level one reflective Lisp interpreter that has direct access to the structure of the level 0 interpreter. Unlike a Pascal interpreter and its associated debugger, however, the ability to move freely between different interpreter levels in a reflective language makes it practically impossible to compile reflective programs; evaluating such a program essentially requires keeping the compiler resident at runtime to perform dynamic code-generation whenever a reifying operation is executed.

Even though Symmetric Lisp permits the free use of map structures, the behaviour of the underlying interpreter is fixed – one cannot alter the name lookup process or the semantics of map construction (as is possible in a reflective language). The underlying structures that define the Symmetric Lisp interpreter are inaccessible to the user.

On the other hand, the reflective languages of Smith or Friedman and Wand do not allow users to express first-class transparent environments (as is possible in a symmetric language). Environments in these languages are first-class but only insofar as they can be examined by a higher-level interpreter - user expressions cannot examine them nor can they be accessed from within the same level as they are defined. In other words, the distinction between an object that can be examined and an active program is still strictly maintained even in these reflective languages.

1.5.1.3 Pebble

Pebble[15] is a kernel language designed to support the construction and maintenance of large programs in a semantically clean and modular way. It does so by providing a semantics for data types, abstract data types and modules within the framework of the second-order lambda calculus. Among its many interesting features (*e.g.*, dependent types, polymorphism, types as values, etc.), Pebble also treats bindings (and naming environments) as values.

A Pebble binding **x~3** binds the value of the name **x** to 3. The role of bindings in Pebble is to provide a basis for a semantics of modules given in terms of functions that map bindings to bindings. Pebble modules are treated as functions that map environments containing the bindings of module interfaces to environments containing the operations the module defines.

Because Symmetric Lisp maps are first-class environments, they can be used to implement Pebble-style modules[6]. Pebble bindings and declarations, however, are not data structures in the Symmetric Lisp sense - they cannot be selected out of the environment in which they are defined nor can the user iterate over the bindings declared within them.

1.5.2 Parallelism

1.5.2.1 Expression-Level Parallelism

The parallel evaluation semantics of Symmetric Lisp maps is similar to the fine-grained parallel evaluation semantics most commonly found in functional languages[12, 43]. All map elements

[6]Pebble modules are also strongly-typed. We discuss a type-inference algorithm for Symmetric Lisp in Chapter 6.

evaluate concurrently unless otherwise constrained by data or control dependencies.

Symmetric Lisp's non-strict evaluation semantics is also similar to the evaluation semantics found in non-strict, lenient languages such as VimVal[26] or Id[53] and distinguishes it from non-strict, lazy languages such as Miranda[65] or SASL[66].

1.5.2.2 Explicitly Parallel Languages

Programs written in explicitly parallel languages are structured as a collection of communicating sequential processes with inter-process communication patterns explicitly specified by the programmer. Because Symmetric Lisp maps serve the role of both program and data structure, they can be used to implement shared resource managers that may be accessed by many concurrently evaluating processes executing within other maps.

A symmetric language map acts as a *distributed data structure* – its components may be selected and examined by many processes simultaneously. Most explicitly, parallel languages (*e.g.*, Ada[3], CSP[42] or Qlisp[34]) do not provide direct support for this kind of structure. Instead, they provide manager or monitor processes, such as an Ada entry procedure, a Qlisp process-closure or a CSP-Occam output statement. The consumer calls the manager to request or to update some item and the manager returns the item or updates it as requested; the consumer waits until the request is satisfied. Remote procedure call schemes require an extra and logically unnecessary inter-process transaction, between the consumer and manager, compared to distributed data structure schemes.

Among explicitly parallel languages, Symmetric Lisp is most closely related to Linda[35]. Linda processes communicate via a globally-shared collection of ordered tuples called a tuple-space. There are operations to add, remove, and read tuples from this space. A tuple is a distributed structure; once added into tuple space, any process that wishes to read it may do so. If no tuple is available, the process suspends until one is, then proceeds. Tuples may be added to tuple-space even if unevaluated; such tuples are Linda's counterpart to Symmetric Lisp's non-strict map structures. Even though the evaluation semantics of Linda tuples match well with that of Symmetric Lisp's maps, Linda is based on a very different programming model. Linda tuples do not define naming environments. All tuples reside in a flat namespace and Linda program structures, unlike Symmetric Lisp programs represented as maps, are not represented as tuples of a particular structure. Linda processes reside in a namespace different from the tuple-space that they build and access.

MultiLisp[40] is a parallel language derived from Scheme by the addition of a special non-strict constructor called a **future**[7]. The expression, (**future** X) where X is an arbitrary expression, creates a task to evaluate X and also creates an object known as a *future* to hold the value of X eventually. The result of a **future** expression is the *future* object; any expression that attempts to read the value before it is produced blocks until it becomes available. In this sense, futures are similar to early-completion structures implemented in VimVal and are a restricted form of I-structures[7] found in Id or empty regions constructible in Symmetric Lisp. Insofar as parallelism is concerned, a Symmetric Lisp map could presumably be implemented as a MultiLisp **letrec** expression in which each binding expression is wrapped inside its own future. A **letrec** doesn't define a data structure, though, and such a transformation would need to be greatly extended if it were to be faithful to the semantics of map evaluation as dictated by the symmetric programming model.

1.6 Overview of the Thesis

Chapter 2 begins with an informal description of Symmetric Lisp and presents simple examples of map expressions and how they may be built and used. Chapter 3 presents a formal operational semantics of the language using Plotkin-style rewrite rules.

Chapter 4 gives examples of how to express program and data constructs found in many diverse existing languages in Symmetric Lisp. The main goal in this chapter is to show that the symmetric programming model is a good thought-tool for reasoning about programs as data objects and, conversely, for reasoning about data objects as processes. The chapter begins by showing how to implement common program structures found in existing languages - local naming blocks, cobegin-coend statements, lazy structures, Simula and Smalltalk classes, inheritance, etc. - purely in terms of selection and definition of Symmetric Lisp map structures. Some new program structures are also developed that are not easily supported in other languages. These include parallel knowledge daemons and programs based on dynamic process streams. The realization of these programs crucially depend on the uniformity of program and data and

[7]Note that Qlisp also supports futures.

the ability to examine programs (or processes) as they are under evaluation. The latter part of the chapter discusses the use of maps as distributed data structures and the role of empty regions as an explicit synchronization device. Examples of a mutually-recursive stream program, a dataflow-based simulator and resource managers[5] are given to illustrate the utility of a symmetric language in expressing concurrency. All the example programs given in this chapter have been tested using a logically concurrent interpreter for the language implemented in Common Lisp on a TI-Explorer.

Chapter 5 discusses the Symmetric Lisp programming environment. First-class parallel environments make Symmetric Lisp suitable as a base language for a monolingual, parallel language-based computer system. The structure of a simple file-system written in Symmetric Lisp is sketched and some examples of daemon processes that can watch environments or streams for interesting developments are given. These daemon processes can be written and installed in a user's environment directly; no interaction with system-supplied utilities is necessary.

The viability of any Symmetric Lisp implementation depends on how effectively the compiler can translate symbolic name references to target language addresses. The problem is an important one in Symmetric Lisp because the evaluation environment of any expression can be dynamically altered by enclosing it within a user-specified environment structure. Chapter 6 shows how this problem can be solved using abstract interpretation; by making the "type" of an expression contain information about the names it defines and uses, it is shown how a compiler can infer the proper evaluation environment of an expression.

The fine grained parallel evaluation semantics of maps coupled with the model's block-on-uncomputed-regions rule makes Symmetric Lisp suitable for implementation on a general purpose dataflow machine. In Chapter 7, we present a source-to-source translation of Symmetric Lisp programs into the high-level dataflow programming language Id[53]. Given a translation scheme from Id into the base language representation of dataflow graphs, it becomes straightforward to understand how Symmetric Lisp programs might be represented as dataflow graphs and implemented on a dataflow machine. We do not undertake a discussion of architectural issues such as resource management, code mapping, etc., since these issues pertain to dataflow systems in general, and are not specific to any particular Symmetric Lisp implementation.

Chapter 8 gives conclusions.

Chapter 2

Symmetric Lisp

The last chapter presented an abstract description of the symmetric programming model. In this chapter, we make the abstract description concrete by informally presenting the syntax and semantics of Symmetric Lisp. We give some simple examples of Symmetric Lisp programs and describe the mechanism by which maps are constructed and manipulated. In Section 2.2, we use the operators defined in the earlier parts of the chapter to build an important abstraction (called an *open-map*) that is used extensively in later chapters of the thesis.

There are two appendices. The first gives a quick reference guide to the operators and terms defined in the chapter; the second presents a formal BNF description of the grammar.

2.1 Description of the Language

Expressions in Symmetric Lisp, for the most part, follow Common Lisp syntax. (The main exceptions are name-binding expressions and various selection operations.) The language supports all standard arithmetic, boolean, conditional and string operations found in Common Lisp.

2.1.1 The Program Structure

A Symmetric Lisp *program* is an unevaluated map form which, upon evaluation, becomes a map value; a map value, in other words, is a map expression in normal form. A map form is *not* an expression that yields another map as result. Maps have *spatial identity* - they are evaluated in place, without copying, and when fully evaluated become map values.

A map consists of a number of expressions called *regions.* Each expression may have a name; an expression with a name is referred to as a *named expression.* Giving a name (which must be a symbol) to an expression binds the name to the result yielded by the expression; this binding is visible to any expression which has access to the map. A binding is written as:

```
name : e
```

This binding associates `name` with the value which expression `e` denotes. Within a given map, all names must be unique.

2.1.2 The Map Evaluation Rule

A map expression is evaluated in two steps. First, all region-defining names are recorded as elements of the environment to be defined by this map. After this step, the map structure is accessible to any expression that requires it. Its evaluation is only complete, however, when all expressions in all of its regions have been evaluated. These expressions are evaluated simultaneously; if an expression requires the value of some binding and that value is still being computed, evaluation of the expression ***blocks*** until the required value becomes available. The k^{th} element of an evaluated map holds the result yielded by evaluation of the k^{th} expression in the original. A named element in the original retains the same name in the evaluated version. Thus, the map expression

```
(map
   x : (+ 1 1)
   y : (+ x 10)
   (+ y 10))
```

defines a three-region map; the first region is named `x` and contains the expression `(+ 1 1)`. The second region in the map is named `y` and contains the expression `(+ x 10)`. The third region is unnamed and contains the expression `(+ y 10)`.

This map, when evaluated, becomes

```
(map
   x : 2
   y : 12
   22)
```

It's natural to allow access to maps before they are fully evaluated, because their structure is known by the end of the evaluation phase in which names are established. Both this non-

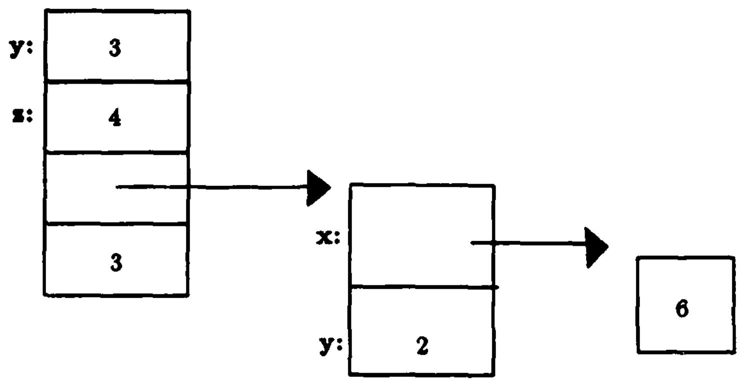

Figure 2 : Structure of a Map

strict access rule and the synchronization-on-uncomputed-values rule resemble procedures that govern functional data structures implemented using early-completion structures[26].

Every map has an associated unique system-provided address. The `apply-env` operator returns the address associated with the map within which it is evaluated[1].

A map defines a local lexical scope to which the expressions within it refer during their evaluation. Thus, when

```
(map
   y : 3
   z : 4
   (map
      x : (map
             (+ z y))
      y : 2)
   y)
```

is evaluated, it becomes

```
(map
   y : 3
   z : 4
   (map
      x : (map 6)
      y : 2)
   3)
```

Figure 2 shows the representation of this map object.

[1]Such an operator is implemented in some object-oriented languages as a special variable called *self*[38] or *this*[23].

In the above example, references to **z** and **y** made during the evaluation of the **+** operation get resolved by first looking for bindings for these names in the current map, (which defines no bindings), then in the map in which this map resides (which defines a binding for **y**) and then in the map in which this enclosing map resides (which defines a binding for **z**).

Every expression has an associated evaluation environment that specifies where it should retrieve its free names. Although the terms "map" and "environment" are used interchangeably throughout the thesis, the usage is actually a bit imprecise: a free-standing Symmetric Lisp map is simply a frame of bindings and values. The name-lookup rule described above captures the notion of an evaluation and naming environment, but is meaningful only when the expressions found in map regions are evaluated relative to their lexically enclosing environment. The interpreter maintains the information regarding the evaluation environment of expressions.

One can ask for the number of elements in a given map using the **msize** function — (**msize M**) returns the number of regions in map **M**.

2.1.2.1 Constraining Evaluation Order

All elements of a regular map evaluate in parallel with no pre-specified evaluation order. Elements of a **seqmap** (sequential map), on the other hand, evaluate in a particular order starting with the first (leftmost) region and proceeding to the right. After all elements in the first region have finished evaluating, the evaluation of elements in the next region can begin. A fully evaluated seqmap is identical to a regular map; the result of a seqmap evaluation is an ordinary map value.

One can define name bindings in a seqmap. All names declared within a seqmap are defined simultaneously, before the evaluation of the first element commences. Since all the names found in a seqmap are known before expression evaluation actually begins, mutually recursive expressions or forward references occuring in a seqmap may lead to deadlock. Consider, for example, the following fragment:

```
(seqmap
    x : y
    y : (+ 2 3))
```

Both **x** and **y** are recorded as part of the environment defined by the seqmap before the evaluation of the first expression begins. Because of the serial evaluation rule, however, the binding of name **y** to 1 will never be installed because the expression **y** in the binding, "**x : y**" must first resolve to a value; since the expression responsible for producing this value, `y : (+ 2 3)`, can execute only after the first expression terminates, a deadlock situation exists.

What does it mean for an element to "finish evaluating"? Informally, an element in region r is said to have finished evaluating if outside expressions do not block in attempting to access r's contents. Thus, if an element is a primitive expression such as an arithmetic operation found in region r, evaluation of region $r + 1$ can begin only after the operation yields a value. The tricky part in defining what we mean by "finish" has to do with reconciling this term with the non-strict evaluation rule for maps. We said earlier that a map is available for inspection once all the names it declares have been recorded as part of its evaluation environment. We did not require that all subexpressions found in the map evaluate to values before the map can be accessed. Although the map expression (in a sense) has finished evaluating (insofar as the names it defines have been recorded and an internal representation of it has been created by the interpreter), evaluation of its subexpressions need not have completed. This non-strict evaluation rule means that a map expression that happens to be the i^{th} element in a seqmap can have its elements still evaluate in parallel with the seqmap's $i + 1^{st}$ element; the seqmap construct does evaluate elements in a particular order, but does not ensure that its elements evaluate sequentially if some of them have a non-strict evaluation semantics.

The following definitions make this informal description precise:

Definition 2.1 *A* value *is either a constant (*e.g.*, an integer, boolean, string or function) or a map all of whose elements are values. An* object *is either a constant or a map for which an internal representation exists.*

Definition 2.2 *An expression e has* finished evaluating *when it yields an object as its result.*

This weak constraint on evaluation order implies that the deadlock situation illustrated above may be avoided through the judicious use of map expressions; the above example would not deadlock if the first binding were instead written:

```
    x : (map y)
```

(Note that this transformation is valid only if no other expression accesses **x**.) In the presence of side-effecting operations, seqmap elements that execute after a map expression may interfere with the evaluation of component expressions in that map. The potential for interference between successive seqmap elements prohibits one from asserting that a seqmap simply mimics a sequential interpretation of a regular map.

Consider the following program fragment:

```
(seqmap
   x : 0
   f : (map
          ...
          (set x 1)   ;;; set  is an assignment operator
          ... )
   (set x 2))
```

Evaluating this expression first causes the names x and f to be recorded as part of the evaluation environment for the seqmap. After the constant 0 is bound to x, the map expression associated with f is evaluated. Because maps are non-strict, it is not possible to determine the order in which the multiple assignments to x found in the program take place. Unlike a serial execution semantics that would order the evaluation of the map expression containing the assignment (set x 1) with the evaluation of the second assignment (set x 2), the semantics of seqmap makes it possible to interleave the second assignment with the evaluation of other elements in the inner map. Since it is not possible to guarantee the absence of any overlap in the execution of the map bound to f and the last assignment operation, expressions in a seqmap are *not* serializable in the database sense[30]. The main purpose of the seqmap constructor is to ensure that some strict operations (*e.g.,* assignment) take place before other strict (or non-strict) operations. Explicit locking mechanisms are used to implement true serializable execution; these locks prevent evaluating expressions from observing or interfering with the intermediate states of a shared object being manipulated by another activity. The means by which locks are built is described in Section 2.1.4.

2.1.3 Fixing the Evaluation Environment

Environments, being first-class objects, may be bound to names. If an environment is bound to a name Q — Q: (**map** ...) — one can evaluate an expression E using the names defined in Q by writing (**with** Q E); such an expression is referred to as a *scope-expression*, Q is referred to as an *environment-specifier* and E is known as the *scope-body*.

Suppose that Q is the following map:

Q : (**map** R_1 R_2 ... R_n)

To evaluate the above scope-expression, the interpreter evaluates E consulting the bindings defined by `Q` for the meaning of any free name it encounters. If a free name occurring in the body is bound to a region in `Q`, the name acquires the value of that region. Free names in E not bound in any of `Q`'s regions are looked up within the map immediately enclosing the **`with`** expression and so on, as per the normal evaluation rules. (Although the notion of a free name should be clear to the reader, a precise definition is given on page 59.)

The map object yielded by the environment-specifier is viewed as a naming environment. The bindings defined in each named region of the map contribute to this environment; a map defines a naming environment that associates every named region in the map with the value of the region to which that name is bound.

Symmetric Lisp follows a Lisp-like call-by-sharing policy – maps are shared (not copied) among the expressions that access them while scalars are copied and not shared. Thus, the evaluation of `Q` in the above **`with`** expression returns a *reference* to (not a new copy of) the map to which `Q` is bound; E can, therefore, cause side-effects in `Q`. Note that it is not necessary for `Q` to be a simple name – any expression which yields a map as its result is acceptable as the first argument to the **`with`** form.

The value yielded by E is returned as the value of the **`with`** form. Thus,

```
(map
   y : 2
   x : 3
   Q : (map y : 1)
   (with Q (+ x y)))
```

becomes

```
(map
   y : 2
   x : 3
   Q : (map y : 1)
   4)
```

$Q.E$ is an abbreviation for the expression

(**with** Q E)

The dot abbreviation is right-associative: $Q.R.E$ is an abbreviation for

(**with** Q (**with** R E))

It is sometimes useful to restrict the visibility of names defined in a map especially if the map's name-bindings can be made visible to expressions not necessarily defined within it (as is possible using the **with** form). The binding:

```
priv name : e
```

when evaluated in map **M** hides **name** from any expression that uses **M**. In other words, **name** is only visible to expressions defined within **M** and to the expressions found in sub-maps directly enclosed by **M**; if **M** is used as the source environment in a **with** expression, **name** is not made visible to the expression enclosed by the **with**. A discussion of how to use this style of information-hiding to implement a simple form of data abstraction is given in Chapter 4.

2.1.4 Locking an Environment

Because maps have the behaviour of ordinary data structures, it is useful to think of them as objects whose elements may be seized by an evaluating expression and released when the expression finishes. By allowing expressions to gain exclusive access to an environment's elements, one can build programs in which concurrently executing expressions *do* exhibit serializable behaviour — a feature not captured by the semantics of **seqmap**.

A **lock** is a special constant. A binding,

```
x : lock
```

defines **x** to be a lock; the value of **x** is the keyword **lock**. The only operation one can perform on a lock is to seize it.

The expression

(holding e_1 e_2**)**

first evaluates expression e_1 to yield a lock object l. If no other expression currently holds l, then the lock is given to e_2. Once a lock is obtained, evaluation of e_2 proceeds as normal. Any other expression that executes a **holding** operation using l blocks until e_2 yields a value; once e_2 completes, the lock is released[2]. The value of the **holding** expression is the value yielded by e_2. If, because of the evaluation of another **holding** expression, lock l has already been seized

[2]This means that if e_2 is a map expression, the lock is released only when all of e_2's elements become values. This restriction is severe but necessary in order to provide serializability. If e_2 were a map expression, and the lock was released as soon as e_2 became a map *object*, we cannot rule out the possibility of interference between an assignment performed by an sub-expression of e_2 on the structure protected by e_1 and other outside concurrently executing assignments.

by some other expression, e_2 blocks until that expression finishes and the lock is released. If there are several expressions waiting for the same lock, the expression which gets the lock next is chosen arbitrarily. The implementation is guaranteed to be starvation-free insofar as all expressions waiting for a lock ultimately get it assuming that the lock is eventually released by every expression which seizes it.

If `M` is the map

```
(map
   incr : lock
   val : 0)
```

then the expression,

```
(holding M.incr (set M.val (1+ M.val)))
```

first seizes the lock bound to `incr` and subsequently increments the value of `val`. Assuming that all expressions accessing this map obey the discipline of acquiring the lock before updating `val`, this map could be used as an atomic counter. (The assignment evaluates its first argument to get an address and replaces its contents of the address with the value yielded by evaluation of its second argument.)

As in any concurrent system implementing locks, great care must be taken to avoid entering into deadlock situations. Consider the following fragment:

```
(map
   x : (+ (holding Q.a y) (holding R.b e2))
   y : (+ (holding Q.a e3) (holding R.b x)))
```

where environment `Q` defines lock `a` and environment `R` defines lock `b`. Suppose that the first expression acquires lock `a` and the second expression acquires lock `b`. Lock `a` will not be released until `y` becomes bound to a value, but `y` cannot become bound to a value until lock `a` is released. Note that locks prevent both readers as well writers from accessing the map. Excessive use of locks may, therefore, significantly reduce concurrency. Symmetric Lisp encourages a functional programming style; locks should be used sparingly, and only in cases where a shared resource must be updated.

2.1.5 Selection

Three kinds of selection are possible over maps. One can ask for arbitrary sub-maps of a given map either by selecting a subset of the elements of a map by name, or by selecting a subset of the elements defined in a map by position, or by asking for the value of the last region in a map. **select** implements the first, **index** implements the second and **mlast** implements the third.

select is a special form that, given a map, **M**, and a collection of names, n_1, n_2, ..., n_k, returns a k-element map that contains a binding for each of the k names; the binding-value for the i^{th} name is the value of the region to which this name is bound in **M**. Any argument name not found in the specified **map** is bound to a special error value, **error**; the value of any expression which accesses an error element or contains an error element is also **error**[3].

Thus if Q is the map:

```
(map
   foo : v1
   bar : v2
   bam : v3)
```

then (**select** Q **foo bar bam**) returns:

```
(map
   foo : v1
   bar : v2
   bam : v3)
```

Because of Symmetric Lisp's call-by-sharing semantics, if v_1 in the original map itself references a map structure, the name **foo** in the map returned by the **select** expression is bound to the same environment address. Thus, any change to made to the map referred to by v_1 made in Q will be visible in the projected map and vise-versa. Because scalar values do not have environment addresses, original and projected maps never share scalars.

(Note that the order of the name arguments in the call are significant – (**select** Q **foo bam**) is *not* the same as (**select** Q **bam foo**) since elements in the projected maps can be selected by position as well as by name.)

index is a special form that given a map and a collection of indices returns a new map containing the contents of the regions associated with those indices in the region that the indices denote.

Thus, given Q above, (**index** Q 1 2) yields:

```
(map
```

[3]Once a name is bound to **error**, it cannot be subsequently changed (via **set**) to a non-error value. This approach to propagating error values throughout a program is similar to the method of error handling found in some dataflow languages[70].

```
    v1
    v2)
```

If an argument index is given that is greater than the number of regions in the map (or less than 1), it is an error.

Argument indices may be any expression that evaluates to an integer. Like **select**, the non-scalar elements of the **map** returned by **index** are shared with the **map** from which they were extracted.

Finally, one can ask for the *value* of the last region of a map using the **mlast** operator. Thus, (**mlast** Q) yields v_3. If Q is an empty map, **mlast** returns **error**. Note that **mlast** cannot be implemented just in terms of **index** and **msize** since **index** always returns a map; **index** cannot be used to just extract an element from a map.

Given **mlast** and the two projection operators, one can define arbitrarily complex selector functions. For example, one can define the Symmetric Lisp equivalent of the Lisp **aref** function, **mnth** on M as follows:

```
(mlast ( index  M i))
```

for some map M and integer-expression i[4]. It is convenient to use the abbreviated notation, M[i] to denote the above expression; in general, the expression M[i,j,...,k] is shorthand for:

```
(mnth ... (mnth (mnth M i) j) ... k)
```

The ability to extract elements of a map by position or name undermines to some degree the protection afforded by **priv** bindings; even if a name is defined to be private, one can access the contents of the region to which it is bound via appropriate selector operators. Symmetric Lisp does not impose visibility constraints on regions, but only on names: ensuring that a region bound to a **priv** name is not accessed by its position in the map is the responsibility of the user; Symmetric Lisp does not enforce such constraints.

2.1.6 Environment Abstractions

Symmetric Lisp programmers can abstract over environments or expressions to create functions. Both kinds of abstraction are closely related, and can be thought of as simple mechanisms for defining and manipulating **maps** of a specified structure. Environment abstractions are described here and procedural abstractions are discussed in the section following.

[4]The non-recursive definition of maps makes them more akin to arrays or vectors than lists.

Symmetric Lisp allows the programmer to build environment-creating *templates* using the **kappa** form. Applying a Symmetric Lisp **kappa** of k argument names to k actuals causes the evaluation of a map in which each argument name is bound to the value yielded by evaluation of the actual and whose last element is the body of the abstraction. Kappas are lexically-scoped: their evaluation environment is fixed at the point of definition; evaluation of a kappa is immune from the bindings defined in the apply-time environment. Kappas may be higher-order, *i.e.*, they may be returned as the result of an application, built into maps, etc.

The **kappa**-application process works as follows. (Assume that free names in the kappa body have been suitably translated to reference their binding in the kappa's define-time environment.) The expression

```
((kappa (var1 var2 ...  varn)
      body)
 exp1 exp2 ... expn)
```

expands to the map

```
(map  < actuals>).(map < formals>  body)
```

where (map <*actuals*>) is

```
(map
   new-id1 : exp1
   new-id2 : exp2
           ...
   new-idn  : expn)
```

(the *new-id*'s are new identifiers) and (**map** <*formals*> *body*) is

```
(map
   var1 : new-id1
   var2 : new-id2
          ...
   varn :  new-idn
   body)
```

for the same new-id's used above. The bindings of a formal to the actual in the map generated by kappa application may be declared private if the formal is prefixed in the kappa declaration by priv. Thus, the definition

```
(kappa ((priv var1) var2) <body>)
```

declares the binding of $\texttt{var}_1$ in the map generated by kappa application to be private.

This map is an ordinary Symmetric Lisp expression and is evaluated by the standard rules: the $\texttt{exp}_i$ are evaluated in parallel with the body of the kappa ; the scoping rules for maps and the semantics of **with** guarantee that evaluation of the **exp**$_i$ won't pick up the bindings for the **var**$_i$. The value yielded by evaluation of this **with** expression is the map containing the bindings for each of the formals whose element is the value of **body**.

(For brevity, this full expansion is omitted in subsequent examples; the values yielded by the **exp**$_i$ will be used directly in place of **new-id**$_i$ in the inner map.)

Although free names in a kappa are by default evaluated in the kappa's lexical environment, one can explicitly evaluate a free name within the kappa's application environment by wrapping the identifier inside a scope-expression whose source environment is the value of **apply-env**. The expression:

```
(kappa (id1 id2 ... idn)
    ... (apply-env).free-name ...)
```

evaluates **free-name** in the kappa's application environment; unless other free names have their evaluation environment similarly prefixed, they evaluate in the kappa's lexical environment. Examples showing the use of **apply-env** are given in Chapters 4.

Kappas can be used as record generators, analogous to record types in Pascal or **defstructs** in Common Lisp.

Thus, for example, one can write:

```
airship : (kappa (year maxspeed color))
```

If one now applies **airship** to arguments

```
goodyear-blimp1 : (airship 1947 16 "silver")
```

the map

```
goodyear-blimp1 : (map
                      year : 1947
                      maxspeed : 16
                      color : "silver")
```

is returned. The expression **goodyear-blimp1.color** yields "**silver**". Kappa objects can also be used to specify parameterized record types:

```
airship : (kappa (date color filled-with )
              safe : (if (equal filled-with "hydrogen")
                         "nope"
                         "probably"))
```

The invocation

```
hindenburg : (airship 1936 "silver" "hydrogen")
```

yields

```
hindenburg : (map
                  year : 1936
                  color : "silver"
                  filled-with : "hydrogen"
                  safe :  "nope")
```

Other interesting uses of kappa structures are given in Chapter 4.

2.1.7 Procedural Abstractions

Functions in Symmetric Lisp are defined using **lambda** expressions. A lambda expression is structurally identical to a kappa; like kappa objects, free names in a lambda's body are resolved in the function's lexical environment. A lambda, like a kappa, may be higher-order: it may be returned as the result of an application, built into maps, etc.

Applying a lambda of k arguments to k actuals causes the evaluation of a map that is identical to the one given above in the kappa-application case, except for the addition of an **mlast** that is wrapped around the entire expression.

Thus, given:

```
K : (kappa (x y) (+ x y))
L : (lambda (x y) (+ x y))
```

`(K 1 2)` yields

```
(map
   x : 1
   y : 2
   3)
```

and `(L 1 2)` yields the **mlast** of this map – namely 3. Note that the kappa and lambda bodies are implicit maps; if they consist of n separate elements, these n appear as the last n of the invocation map. We can express the expansion rule for lambda application as follows: if f is a lambda form, then (f < *args* >) expands to:

(**mlast** (**map** <*actuals*>).(**map** <*formals*> *body*))

where <*actuals*> and <*formals*> are as defined in the **kappa** application case.

Note that

```
(kappa (x y) (+ x y))
```

is not the same as

```
(lambda (x y)
   (map
      x : x
      y : y
      (+ x y)))
```

Because a map defines a recursive naming environment, the evaluation of expressions x and y in the map defined in the body of the lambda would not terminate since they refer to the name whose binding-value they define. The above kappa structure is, in fact, equivalent to

```
(lambda (f1 f2)
   (map
      x : f1
      y : f2
      (+ x y)))
```

where f1 and f2 are fresh names.

Although there are a number of powerful map-combining forms in the language that can be used to implement simple iteration, general iteration is expressed using tail-recursive functions; there are no special iterative control structures in the language.

It is important to note the conceptual significance of describing the semantics of lambdas and kappas in terms of map abstraction. The basic role of kappas and lambdas is to provide a means by which *names* can be abstracted over maps. Because elements found within a map object may be selected by position, function application in Symmetric Lisp follows trivially from the ability to parameterize over maps – the ability to build a parameterized map via kappa application and the ability to select an element out of any map makes it easy to describe the behaviour of lambda application in terms of map instantiation and selection. Parameterized maps are fundamental to the function application process.

2.1.8 Building Layered Regions

To support both the inheritance of methods in object-style programming and the interpreter's incremental top-level interface, it is important that we be able to supersede bindings in a map in some way. To supersede a binding of a name x in a map M, we build a new map map N with a region named x, and then "layer" this new map onto M. Layering a new map onto an old one causes any name defined in the old map but *re*defined in the new one to have its old definition superseded by the new definition. Thus, if r is a region named x but M already defines a region named x, layering N on top of M returns a map in which x's definition in N supersedes its definition in M. If M does not define a region named x, a map is returned which represents the append of M onto N. Thus, if M contained two regions y and z, the map yielded by the layering operation would consist of regions r, y and z in that order. The layering process may be repeated any number of times. Note that the regions in M and N are themselves unaffected by the layering operation. Examples of layered regions are given in Section 4.1.6.2 and Section 5.1.

The `layer` operator composes maps in this way. Let `M` be

```
(map m₁ : M₁
     m₂ : M₂
      ...
     mⱼ : Mⱼ)
```

and `N` be

```
(map n₁ : N₁
     n₂ : N₂
      ...
     nⱼ : Nₖ)
```

Let $X = \{x_1, x_2, \ldots, x_i\}$ be the set of names that are bound to regions occurring in both `M` and `N`. Then, the expression

```
(layer M N)
```

yields the map:

```
(map L₁ L₂ ... Lⱼ Lⱼ₊₁ ... Lⱼ₊ₖ₋ᵢ)
```

where

- for $1 \leq\ t \leq\ j$, L_t is $m_i : M_i$ if M_i is a region not bound to a name in X and is m_i: (`with` $M\ m_i$) if $n_k = m_i$.
- If N_j is the contents of the left-most region in N not bound to a name in X, then L_{j+1} is the result of evaluating the expression n_j: (`with` $N\ n_j$), L_{j+2} is the contents of the second left-most region in N not bound to a name in X and so on.

Note that the skeleton of the layered map can be returned once the skeleton of the argument maps are known; the value of the regions in the layered-map are determined once the corresponding values in the argument maps become known; thus, `layer` can return a map object even if expressions in its argument maps have not all finished evaluating.

For the purposes of the construction, unnamed regions in the argument maps can be thought of as being bound to some unique name guaranteed not to be defined elsewhere.

`Layer` returns a new map and does not side-effect its arguments. Note that the map returned by the layer operator causes binding-values for names defined in `M` to supersede the binding-values for the same names defined in `N`; the order of the argument maps to layer is, therefore, significant.

As per the semantics of scope-expressions, the map returned by `layer` shares its non-scalar regions with its argument maps; scalar values are copied. The `layer` operation is associative:

(layer R (layer S T)) $\equiv$ (layer (layer R S) T)

`Layer` can be applied to a variable number of arguments; thus,

(layer M_1 M_2 ... M_n) $\equiv$ (layer M_1 (layer M_2 ... (layer M_{n-1} M_n)))

Maps built from the `layer` operation can be used to act as *changing namespaces*; as layer operations are performed, the map's naming environment is changed correspondingly. Because of the non-strict evaluation semantics of maps, we can treat the layering operation as constructing an *evaluation-site* to which new name bindings may be added and old name bindings superseded. The evaluation environment defined by a layered map is determined by the names defined in the regions currently found within the map.

Suppose a map called `M` — `M : (map)` — is created and the following expression is then executed:

```
(map
   M1 : (layer M  (map foobar : (* 2 3)))
   M2 : (layer M1 (map bazball : (* 4 5))))
```

After all expressions have evaluated, `M1` would look like:

```
(map foobar : 6)
```

and `M2` would look like

```
(map
   foobar : 6
   bazball : 20)
```

M2 is a map whose first region contains a binding for **foobar** and whose second region contains a binding for **bazball**; if we now execute:

```
(layer (map foobar : (* 3 4)) M2)
```

the value yielded is the map

```
(map
   foobar : 12
   bazball : 20)
```

The contents of the region named **foobar** supersedes the definition of **foobar** provided earlier. Expressions evaluated in the context of this map will only see the new definition of **foobar**.

2.1.9 Map Generation

The simplest way of generating a map is to use the **iota** primitive function. The expression (**iota** 1 n) for some positive integer n, returns an n-element map whose i^{th} element is i.

Iota builds an integer map using its arguments as the map's lower and upper elements. It is also possible to generate more complex maps from existing ones using the **generate** expression. Elements in the map built by **generate** are computed based on the region's value at the corresponding index in the existing map. The expression:

```
(generate (e M)
    body)
```

where **M** is a n-element map, returns a map, N, of the same dimensions as M in which the j^{th} element N is the result of evaluating *body* in an environment in which **e** is bound to the value of the j^{th} element in **M**. If we assume, for example, that **M** contains n elements, then the above generator is equivalent to the following map expression:

```
(map
  (map i : M[1]).body
  (map i : M[2]).body
     ...
  (map i : M[n]).body)
```

Notice that the evaluation of all n elements in the generated map can proceed in parallel. Notice also that if **i** is not free in **body**, the value of **M[i]** won't be used when **body** is evaluated.

The **generate** operator serves two main roles: first, it provides a way of applying a function across an entire map structure *non-recursively*. For example, if we let (**onto f M**) where **f** is some function and **M** is a map be equivalent to the expression:

```
(generate (e M)
    (f e))
```

then onto applies f to each element in M in parallel and returns a map of the same dimension containing the results of the application.

Thus, given a map:

```
V : (10 10 10 10 10)
```

the expression:

```
(onto (lambda (e) (* 3 e)) V)
```

specifies the computation:

```
(map (* 3 10) (* 3 10) (* 3 10) (* 3 10) (* 3 10))
```

which evaluates to a map containing five 30's. By the ordinary map evaluation rule, the intermediate form calls for all five multiplications to be performed simultaneously. The evaluation of the above onto expression behaves like operators typically found in data parallel languages[46].

The other role of generate has to do with its ability to define general recurrence relations over map structures. Consider the following Id[53] array-comprehension expression that defines a wavefront recurrence:

```
A = {matrix (1,N), (1,N)
    | [1,1] = 1
    | [i,1] = 1             || i <- 2 to N
    | [1,j] = 1             || j <- 2 to N
    | [i,j] = A[i-1,j] +
              A[i-1,j-1] +
              A[i,j-1]      || i <- 2 to N
                             & j <- 2 to N}
```

An equivalent definition can be writt^n in Symmetric Lisp using generators as follows:

```
A : (generate (i (iota 1 N))
       (generate (j (iota 1 N))
         (cond ((and (= i 1) (= j 1)) 1)
               ((and (in? i 2 N) (= j 1)) 1)
               ((and (in? j 2 N) (= i 1)) 1)
               ((and (in? i 2 N) (in? j 2 N))
                (+ A[(1- i),j]  A[(1- i),(1- j)]
                   A[i,(1- j)])))))
```

The computation of all n^2 expressions proceeds in parallel; expressions requiring the value of other elements in A block until a value is produced by the appropriate computation. The `in?` predicate returns true if its first argument lies between the bounds specified by its second and third arguments. Compared to the Id solution, the Symmetric Lisp version is ineffcient, requiring N^2 conditional tests; because the array comprehension statement is implemented using I-structures, it can be implemented using just I-structure array assignment – no conditional tests are necessary. An equally efficient implementation in Symmetric Lisp is given in the next section after a discussion of empty regions.

2.1.10 Empty Regions

A *map with a hole* has an empty or undefined region. A star ($\star$) is used to denote an empty region. Thus the map

```
(map ★)
```

denotes a map with a single region whose contents are still unspecified. One can evaluate and return a `map` with empty regions, but expressions which attempt to access such a region block until the region has been filled with a value.

Empty regions can be filled using the `fill` operator. The `fill` expression has three basic forms:

1. (`fill` *name exp*)
2. (`fill` *map-exp.name exp*)
3. (`fill` *map-exp*[*index*] *exp*)

Regardless of which form is used, `fill` ultimately operates on a map region and a value. In the first form, the region is simply the region denoted by *name* in the lexical environment in which the operator is evaluated. In the second form, `fill` evaluates *map-exp* to get a map object, M; the region denoted by `name` in M is the region to be filled-in; if no binding for *name* is found in any of the regions defined by M, an error is raised. The evaluation of the third form is similar to the second except that instead of a name denoting a region, the third argument is an integer expression that yields an index into M; it is an error if the computed index is greater than the number of regions in M.

`Fill` obeys a partial unification-style semantics – if the contents of the region to be filled is empty, `fill` simply drops the value yielded by evaluation of *exp* into the region and returns the value. If the region is *not* empty, `fill` compares the value of *exp* with the region's contents: if they are both primitive values and equal, or if they both refer to the *same* map or region[5], `fill` simply returns the primitive value or the reference as appropriate. In any other case, `fill` returns an **error** value.

Any expression that was previously suspended waiting for an empty region to be filled can resume execution once a fill operation on that region completes. Thus, given:

```
M : (map
        x : ★
        y : (+ x 1))
```

the evaluation of the expression (`fill M[1] 1`) causes `1` to be substituted for the empty region currently bound to `x`[6].

After the `fill` expression evaluates, the above program would turn into the following data structure:

```
M : (map
        x : 1
        y : 2)
```

Given that maps can have arbitrarily many empty regions, one can build a map of unspecified elements:

```
(generate (i (iota 1 n))
    ★)
```

The resulting map is effectively an uninitialized array; expressions which attempt to access its elements block until the corresponding `fill` operation (which plays the role of an array element assignment here) executes. A map whose elements are either values or empty regions is still considered a value. This is because there is no remaining internal computation to be performed by any of the map's subexpressions; the `fill` operation is an intrusive operation: it alters the map from the outside. This definition of map value some practical benefits as well as we discuss in the next section.

Consider now a reformulation of the wavefront recurrence given in the last section. Using empty-regions, one can devise a more efficient solution that avoids the need to include conditional tests in the body of the generate expression:

[5] *Same* in the Lisp **eq** sense: they must refer to the identical map object, *i.e.*, the l-value of the objects to which they refer must be the same. This restriction is a significant one and can hinder some important optimizations like common subexpression elimination, but makes equality test very inexpensive (since the operation need only compare two addresses for equality).

[6] Note that one could have also written (`fill M.x 1`) to get the same effect

```
A: (mlast
     (map
       (fill A[1,1] 1)
       (generate (i (iota 2 n))
         (fill A[i,1] 1))
       (generate (j (iota 2 n))
         (fill A[1,j] 1))
       (generate (i (iota 2 n))
         (generate (j (iota 2 n))
           (fill A[i,j] (+ A[(1- i),j] A[(1- i),(1- j)]
                            A[i,(1- j)]))))
       (generate (i (iota 1 n))
          (generate (j (iota 1 n))
             ★))))
```

A . initially bound to an n × n map containing only empty regions. The **generate** expressions preceding the last expression in the map that defines A's structure are used to fill in the appropriate entries in the matrix. All the generate expressions execute simultaneously; expressions in a **generate** clause that require other values of A block until the referenced regions are filled in by the appropriate fill expressions.

Empty regions are in many ways similar to I-structures found in Id[53] or logical variables found in logic programming languages[68]; filling a value is equivalent to a (partial) unification operation in which the "★" is treated as an unbound variable.

The fill operator is strictly less powerful than the more general assignment operator **set**. Whereas fill only operates over empty regions, **set** can be used to change the binding-value of any region, empty or otherwise. The subset of Symmetric Lisp that contains fill but not **set** is still determinate, however; because of the unification-semantics of fill, different values cannot be filled-in for the same region. Empty regions and fill combine to provide a greater degree of expressivity (and efficiency) than is possible in a purely functional language. We use **set** only when non-determinism is required and we use fill for applications requiring logical variables.

2.1.10.1 Lenient Evaluation and Empty Regions

The ability to bind regions to holes and the possiblity of cyclic dependencies among expressions implies that it is, in general, not possible to determine a static instruction schedule for an arbitrary Symmetric Lisp expression. As an example, consider the following function:

```
f : (lambda (i j)
       (fill a[0] 0)              ;;;  expression (1)
       (fill a[1] (1+ a[i]))      ;;;  expression (2)
       (fill a[2] (1- a[j]))      ;;;  expression (3)
       a : (generate (i (iota 0 2)) *))
```

Now, consider a schedule in which we choose to evaluate

```
a[1] : (1+ a[i])
```

before

```
a[2] : (1- a[j])
```

This works fine if both `i` and `j` are 0, but fails if `i` happens to be 2 and `j` happens to be 0; a deadlock would occur even though there exists a non-deadlocking schedule, namely one in which expression (3) executes before expression (2). Similarly, if we choose to execute expression (3) before expression (2), we would deadlock if `j` happens to be 1 and `i` happens to be 0. Here also, a non-deadlocking schedule does exist, namely a schedule in which expression (2) executes before expression (3). We state without proof the proposition that it is an undecidable question whether an arbitrary program written in a lenient language that supports holes will terminate (i.e., not deadlock) under some fixed instruction schedule.

2.2 An Example: Multi-Streams

With the operations defined thus far, we now define an abstraction that plays an important role in the following chapters. A *multi-stream* is a distributed data structure[20, 35], a data structure that can be examined and manipulated by many processes simultaneously. It has the behaviour of a parallel queue: producers add elements to the back of the multi-stream (in a single atomic step) and consumers read elements from the head. There may be many consumers reading from the multi-stream concurrently, but the multi-stream semantics guarantee that only one producer can append to a multi-stream at any given instant. A multistream bears some resemblance to a functional language stream [9, 69] except that, unlike conventional streams, it may be appended to by arbitrarily many processes.

A multi-stream is an interprocess communication mechanism. If we view the elements of a map as processes, then we can establish a communication stream among them by having them share access to a single multi-stream. A multi-stream is *not* intended to define a special sort of naming environment; its primary role is to serve as an efficient communication structure between concurrently executing processes. We call a Symmetric Lisp multi-stream an *open-map*.

An *open-map* is defined as follows (we give a side-effecting definition below; an equivalent, but less efficient functional version is easy to derive but omitted here):

```
open-map : (map
              create : (lambda ()
                          (map
                             stream-lock : lock
                             stream : (map ⋆ ⋆)
                             tail : stream))
              attach : (lambda (attach-stream val)
                          new-stream : (map ⋆ ⋆)
                          (fill attach-stream.stream[1] val)
                          (holding attach-stream.stream-lock
                             (seqmap
                                (fill attach-stream.tail[2] new-stream)
                                (set attach-stream.tail new-stream))))
              mcar : (lambda (read-stream)
                        read-stream.stream[1])
              mcdr : (lambda (read-stream)
                        (map
                           stream : read-stream.stream[2])))
```

To create a new open-map, we write: `(open-map.create)`. Elements are attached to the open-map in constant time. Note that `new-map` as well as the initial stream is bound to a map consisting only of two empty regions. Such a map is, nonetheless, considered to be a value since it has no internal subcomputations still in progress. The lock on the open-map instance is held as long as it takes to execute the `fill` and `set` operations in the body of the `holding` expression. The `fill` operation extends the multi-stream and the `set` operation sets the tail to point to the new end of the stream. The `fill` operation that drops the new stream value into `stream[1]` takes place concurrently with the updating of the stream structure. The `attach` function returns only after the updating of the stream takes place.

`mcar` and `mcdr` act like their list counterparts: `mcar` returns the first element of the stream and `mcdr` returns the rest of the stream. An `mcar` or `mcdr` expression that is evaluated before a producer has added an element to the stream blocks until such time as an `attach` operation is performed. Note that the object returned by `mcdr` is *not* an open-map but a map with a binding for the name `stream`; it is thus meaningful to evaluate `(mcar (mcdr M))`, but evaluating `(attach (mcdr M) e)` would yield an error.

2.3 Summary and Appendices

In what sense do the operators presented in this chapter form a "complete" or "sufficient" kernel for Symmetric Lisp? Recall that Symmetric Lisp is intended to be a language that directly supports the symmetric programming model. The two most important characteristics of the model are (1) that program structures and data structures have a uniform representation and (2) that first-class naming environments are fundamental to the language semantics. Projection operators, scope-expressions, and abstraction operators are manifestations of the second characteristic; generate expressions, position-sensitive operations, and lock objects are an outgrowth of the need to support the first feature.

We can partition all Symmetric Lisp operations into two broad categories: (1) operations that view maps basically as data structures and (2) operations that view maps basically as programs. Many of the operations (especially in the first category), individually considered, are not particularly unique to Symmetric Lisp; in fact, most of them exist (albeit in different forms) in other languages. The novelty of Symmetric Lisp is not in the collection of operations it supports, but in the way these operators interact with one another in the context of a programming model that enforces no distinction between the program and data objects they manipulate.

There are three main operator classes under the maps-as-data interpretation:

1. Operators that *generate* maps. The **generate** form acts very much like the list-comprehension form in Miranda[7][65] or the array-comprehension form in Id[53].

2. Operators that *project* names and indices. The **select** primitive views maps as records whose named-fields may be selected and spliced-together to form a new map object; its behaviour is similar to projection operators found in relational database languages[22]. The **index** primitive views maps as arrays whose elements may be selected by position.

3. Operators that treat maps as shared, *lockable* data structures. The **holding** special form when used in conjunction with locks behaves much like a semaphore and is syntactically similar to the *seize* and *mutex* operators found in Argus[47][8].

[7]Miranda is a trademark of Research Software Ltd.

[8]**Holding** differs from *seize* in that it is not possible to temporarily relinquish possession of the lock while the expression enclosed by the **holding** operator is evaluated.

Under the maps-as-programs view, we introduced four significant operator classes:

1. Operators that manipulate maps as *environments*. The `with` operator can be thought of a closure operator that closes an expression with the binding-values of free names in the expression drawn from the environment yielded by evaluation of its first argument.

2. Operators that *build* environments. The `layer` operator composes environments.

3. Operators that *constrain evaluation order*. The `seqmap` construct implements a left-to-right evaluation rule for map expressions.

4. Operators that *abstract* over map expressions. The `kappa` special form abstracts over maps; when combined with an appropriate selection operation, it can be used to implement ordinary lambda abstraction.

Although general arguments have been presented here and in the introduction on the merits of the symmetric programming model, detailed examples justifying these claims are deferred until Chapters 4 and 5. These chapters show how to use maps to subsume both conventional program and data structures found in sequential and parallel languages; novel program constructs are also presented that are unique to the symmetric model. Before giving examples, however, we first develop a formal operational semantics of the language in the next chapter.

Appendix A: Glossary

A table listing all operators presented in this chapter is presented below; next to each operator name is a short description of its function and the page number in which the construct was first introduced.

Primitive Forms: [9]

map Map constructor. Builds a map in which each of its component expressions is associated with a region. (page 27).

seqmap Sequential map. Expressions in a seqmap evaluate in a left-to-right order. (page 30).

with Scope-Expression. Evaluates free names found in its second argument using the name bindings defined by its first. (page 32).

layer Join Operator. Builds a layered map. (page 41).

mlast Returns the value of the last region its its map argument. (page 37).

index Projects a collection of indices onto a map and returns the bindings of those regions in the map that are found in the positions specified. (page 36).

kappa Builds a lexically-scoped map abstraction. (page 38).

set Assignment operation. (page 32).

fill Fills an empty region. (page 46).

if There are two conditional forms in the language: `cond` and `if`; `cond` can be thought of as a macro that is built using the `if` primitive form. (page 45).

holding Evaluates an expression after seizing a lock and releases the lock only after the expression evaluates to a value. (page 34).

Macro Definable Operations[10]

[9] A primitive form is an environment or control-flow related construct with idiosyncratic syntax whose semantics cannot be captured by translation into other primitive forms.

[10] A macro definable operation is an operation that can be expressed by translation into other primitive forms and user-definable functions.

generate Generates a new map by evaluating an expression containing a free variable which is systematically bound to each element in the argument map. (page 44.) `Iota` is a simplified form of **generate** defined such that `(iota m n)` returns

```
(map m m+1 ... n)
```

select Projects a collection of names onto a map and returns the bindings of these projected names in a map. (page 36).

lambda Builds an abstraction that, instead of returning an entire map when applied, returns only the last element. (page 40).

logical Operators Logical operators, *e.g.*, `and`, `or`, `xor` etc. are macros. (page 45).

Primitive Functions

msize Returns the number of elements in a given a map. (page 30).

apply-env Returns the address of the map object in which it is evaluated. (page 39).

Common Lisp Operators All arithmetic and string operations are primitive functions.

Predicates All Symmetric Lisp predicates used in the examples are primitive functions.

Special Symbols

priv Declares a binding to be *private*: no expression found outside the lexical environment defined by the map in which a private name occurs has access to the name's binding-value. (page 34).

lock Yields a lock. When evaluated, it returns a new lock object. (page 34).

error Yields a special error value. (page 36).

⋆ Denotes an empty region. (page 46).

Appendix B: Syntax

The BNF grammar for a subset of Symmetric Lisp is shown below. Terminal and keyword symbols are shown in `type-writer` font, and non-terminals are displayed in roman font. Alternatives are separated by double vertical bars (|). Optional expressions are placed in angle brackets, < ... >. The form $\{Exp\}^*$ indicates that Exp may be repeated zero or more times, the form $\{Exp\}^+$ indicates that Exp may be repeated one or more times:

```
Program         ::= (map {<<priv> Id :> Region}*)
Region          ::= Primitive-Form | Abstraction | CL-Form | ⋆ | lock

Primitive-Form  ::= Basic-Exp | Map-Constructor
Basic-Exp       ::= Id | Scope-Exp | Application | Assignment | Select | Conditional
Map-Constructor ::= Program | Generator | Projector

Id              ::= a sequence of alphanumeric characters
Generator       ::= (generate (Id Primitive-Form) Region) | (iota Basic-Exp Basic-Exp) |
                    (layer Primitive-Form Primitive-Form)
Projector       ::= (select Primitive-Form {Id}+) |
                    (index Primitive-Form {Basic-Exp | Integer}+) |

Select          ::= (mlast Primitive-Form) | Primitive-Form[{Selector,}* Selector]
Selector        ::= Basic-Exp | Integer

Scope-Exp       ::= (with Primitive-Form Region) | Primitive-Form.Region
Abstraction     ::= (kappa ({(priv Id) | Id}*) {Region}*) |
                    (lambda ({(priv Id) | Id}*) {Region}*)
Application     ::= (Function {Region}*)
Function        ::= Basic-Exp | Abstraction

Conditional     ::= (cond {(Test Region)}+) |
                    (if Test Region <Region>)
Test            ::= Basic-Exp | Boolean
Assignment      ::= (set Id Region) | (set Primitive-Form.Id Region) |
                    (set Primitive-Form[Selector] Region)
                    (fill Id Region) | (fill Primitive-Form.Id Region) |
                    (fill Primitive-Form[Selector] Region)
Cl-Form         ::= Arithmetic Operations, Boolean Operations (including keywords t and nil)
                    and String Operations
```

Chapter 3

An Operational Semantics

This chapter gives an operational semantics for a significant subset of the language described in the last chapter. We present the semantics of all primitive forms except **holding**, **layer** and **index**, *i.e.*, we give the semantics for **map**, **seqmap**, **with**, **mlast**, **kappa**, **set**, **fill** and **if**. We also do not consider the semantics of **priv** names; here again, incorporating private names into the semantics is conceptually straightforward, but its inclusion complicates the basic name-lookup rules.

The semantics is defined by a collection of *rewrite* rules that describe the transformations to an expression performed by an abstract Symmetric Lisp interpreter. Our rewrite rules differ from more traditional *term rewriting systems*[13, 45] in two important respects:

1. The rewrite-rules do not reflect a semantics based on a substitution model; maps and regions are always shared, never copied. Consequently, the semantics includes meta-functions to read and write regions in environments defined in the abstract interpreter state.
2. Regions can be side-effected; regions containing empty holes can be filled-in and regions containing values may have those values overwritten; the reduction strategy used by the interpreter is, therefore, sharply different from common reduction strategies employed for combinatory reduction systems such as the pure λ-calculus.

Standard term-rewriting systems operate directly over terms in the language of interest; no meta-language constructs need to be introduced. The ability to side-effect and share regions requires our rewrite-rule system to introduce a meta-language containing *states* and *addresses*.

Despite the added complexity of the rewrite semantics brought about because of these special features, many of the techniques used to infer important properties[1] of languages definable by

[1]Confluency and termination are examples of two important properties.

standard term-rewriting system can still be employed. The abstract Symmetric Lisp interpreter is a state transition function that, given a reducible expression and a state, returns a new state that reflects the evaluation of this expression. Given an initial state, the value of a Symmetric Lisp program is the final state produced by the interpreter, *i.e.*, the state in which there are no further reducible expressions. (Of course, there are many useful programs for which there is no final state.)

Before presenting the rewrite rules, we first define some basic terms:

Definition 3.1 An *expression* is defined inductively as follows:

- Values (defined below) are expressions.
- Variables (denoted by lower case italics alphanumeric characters) are expressions.
- A kappa or lambda definition is an expression.
- $(f\ a_1\ a_2\ \ldots\ a_n)$ is an expression if f and $a_1, a_2, \ldots, a_n$ are expressions.
- If $e_1, e_2, \ldots\ e_n$ are expressions or name *bindings*[2], then (**map** $e_1\ e_2 \ldots\ e_n$) is an expression as is (**seqmap** $e_1\ e_2\ \ldots\ e_n$).
- If M and q are expressions, then $M.q$ is an expression.
- If e_b,e_t and e_f are expressions, then (**if** $e_b\ e_t\ e_f$) is an expression.
- If M is an expression, then (**mlast** M) is an expression.
- If v is an expression, then (**fill** $x\ v$) is an expression where x is an identifier, as is (**fill** $e_1.x\ v$) if e_1 is an expression, and (**fill** $e_1[e_2]\ v$) if e_1 and e_2 are expressions.
- If v is an expression, then (**set** $x\ v$) is an expression where x is an identifier, as is (**set** $e_1.x\ v$) if e_1 is an expression, and (**set** $e_1[e_2]\ v$) if e_1 and e_2 are expressions.

Definition 3.2 A Symmetric Lisp value is defined as follows:

- Constants, *e.g.*, numbers, booleans, and strings are values.
- The special symbol, $\star$, is a value.
- A **kappa** or **lambda** *closure* (see page 65) is a value.
- A map structure, (**map** $e_1\ e_2\ \ldots\ e_n$), where each of the e_i are values or, if e_i is a name binding – $i : e$ – where e is a value.
- An address (defined below) is a value.

[2]A name binding is of the form, "*id* : *e*" where *e* is an expression and *id* is a name.

Definition 3.3 We say that identifier n is *free* in *exp* if one of the following conditions hold:

- $exp = n$
- $exp = i : e$ and $i \neq n$ and n is free in e.
- $exp =$ (**kappa**(f_1 f_2 ... f_k) e) and $n \neq f_i$, $1 \leq i \leq k$ and n is free in e. (Similarly for a **lambda** abstraction.)
- $exp = (e_1\ e_2\ \ldots\ e_k)$ and n is free in e_1 or e_2 ... or e_k.
- $exp = e_1.e_2$ and n is free in e_1.
- $exp =$ (**map** e_1 e_2 ... e_k) and $e_i \neq n : e$ for all i, $1 \leq i \leq k$ and n is free in e_1 or e_2 ... or e_k. (The same holds true for a **seqmap** expression.)

3.1 The Structure of the State

A Symmetric Lisp *state* consists of a finite set of *environments*,

$$\{\rho_1 \rho_2 \cdots \rho_k\}$$

where ρ_i represents an *address* for an environment. An environment is a pair:

$$< parent, map >$$

where *map* is an environment address and *parent* is the environment address of the *map*'s parent environment in the state. The name-lookup rule described in Chapter 2 captures the notion of an evaluation and naming environment, but is meaningful only when maps are defined relative to their lexically enclosing parent; the *parent* field in the environment structure is used to specify the evaluation environment of a map.

If an environment has address ρ_i, then its *map* field has address $M(\rho_i)$ and its parent field's address is denoted as $P(\rho_i)$. The meta-level expression, "$P^j(\rho_i)$", evaluates to the address of the j^{th} outermost environment of the environment with address ρ_i; as the base case, $P^0(\rho_i) = \rho_i$. Thus, if $P^i(\rho) = \rho_i$, then the bindings defined by the map with address $M(\rho_i)$ are the bindings found in the i^{th} outermost evaluation environment of the map whose environment address is ρ. If ρ has no parent, then $P(\rho) = \phi$ where ϕ is a unique address that references the "initial" or "empty" environment.

The meta-level semantic function *Contents* is used to manage access to the global state. Given an address α, *Contents* returns the object referred to by α in the current state. The expression

Contents$(M(\alpha))[i]$ returns the value of the i^{th} region in the map whose address is $M(\alpha)$[3]; if the region is named, *Contents* returns just the binding-value and not the entire binding.

The *Size* function, given an environment address ρ_q returns the number of elements in the map addressed by $M(\rho_q)$. In addition to these functions, there are also predicates to test whether an expression is a value, whether a value is an address, etc.

The *initial* state consists of an environment containing a single map expression and ϕ; the parent environment of this map expression is ϕ. The Symmetric Lisp interpreter evaluates this map expression, producing a new state that reflects the evaluation of this expression. A state whose environments contain only values is called a *final* state.

3.2 The Structure of a Rewrite-Rule

It is convenient to describe the behaviour of the abstract interpreter in terms of a collection of *rewrite rules.* Each rewrite rule describes the new state produced by the interpreter given an expression and the current state. When considered collectively, the rewrite rules define the operational semantics for the language. The function *Eval*, representing the interpreter, implements the rewrite rules given below; its domain equation is given as

$$Exp \times State \rightarrow State$$

A *reducible expression* is tagged in our abstract machine with the symbol "?"; an expression tagged with a "?" is said to be *marked for evaluation.* Thus, to reduce the expression (+ 2 1), we denote it in our abstract machine as: [(+ 2 1), ?]. Every rewrite rule either reduces an expression to a value or tags the expression's subexpressions with a "?", thereby making them eligible to be reduced. The evaluation environment of a reducible expression can be explicitly specified. The meta-expression "$[\rho \downarrow \; e]$" is to be read, "Expression e is to be evaluated in environment ρ"; the meta-expression "$[\rho \downarrow \; e, ?]$" indicates that e is a reducible expression that is to be evaluated in environment ρ.

The notation:

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ [e,?]\ldots]\ldots\rho_n\}}{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ e'\ldots]\ldots\rho_n\}}$$

[3]Although *Contents* is evaluated relative to a particular state, we omit including the state if it is obvious from context.

is used to describe a rewrite-rule that denotes the sentence, "Substitute for expression e found in environment ρ_k in state $\{\rho_1\rho_2\dots\rho_k\dots\rho_n\}$ the expression e'."

A single reduction may change several regions or may introduce new environments; unlike conventional term-rewriting systems, a rewrite-rule can, in addition to reducing the current reducible expression, change the contents of other regions or add new environments to the global state. Thus, for example, the rewrite-rule:

$$\frac{\{\rho_1\ \rho_2\ \dots\ \rho_j[\overbrace{\dots\ e_1}^{r}\dots]\dots\ \rho_k[\dots\ [e,?]\dots]\dots\ \rho_n\}}{\{\rho_1\ \rho_2\ \dots\ \rho_j[\overbrace{\dots\ e'}^{r}\dots]\dots\ \rho_k[\dots\ e'\dots]\dots\ \rho_n\}}$$

specifies that, in addition to substituting e' for e, the contents of the r^{th} region in the map whose address is $M(\rho_j)$ (here referred to as e_1) also is to be changed to contain e'.

3.2.1 Constants

The evaluation of a constant expression yields a constant value. Thus, if v is some constant expression, then

$$\frac{\{\rho_1\ \rho_2\ \dots\ \rho_k[\dots\ [\rho\downarrow\ v,?]\dots]\dots\ \rho_n\}}{\{\rho_1\ \rho_2\ \dots\ \rho_k[\dots\ v\dots]\dots\ \rho_n\}}$$

Note that an empty region, denoted by a $\star$, is also considered a value. Because a $\star$ is considered a value, a map value may also contain empty regions. Empty regions do impose some restrictions on how the interpreter may reduce expressions – no name expression bound to an empty region, for example, can be selected for reduction; similarly, no selector expression that selects an empty region can be fully reduced nor can a **seqmap** expression whose component element references an empty region.

3.2.2 Name Resolution

The value of a name is determined by looking for the appropriate binding in the closest enclosing **map** referenced by the environment within which the identifier is to be evaluated. Formally, let $e = [\rho_j \downarrow\ id,?]$, $e' = id : v$, v a non-$\star$ value such that e' occurs in the map whose address is $M(\rho_h)$, and $\rho_h = P^m(\rho_j), m \geq 0$. Suppose, furthermore, that there does not exist $f < m$ such that $M(P^f(\rho_j))$ is the address of a map that defines a region named id. Then

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_h[\overbrace{\ldots\ e'\ldots}^{n}]\ldots\ \rho_k[\ldots\ e\ldots]\ldots\ \rho_n\}}{\{\rho_1\ \rho_2\ \ldots\ \rho_h[\overbrace{\ldots\ e'\ldots}^{n}]\ldots\ \rho_k[\ldots\ v\ldots]\ldots\ \rho_n\}}$$

A name expression resolves into the value to which the name is bound. Note that a variable reference is reducible *only* after the name becomes bound to a value; references to a name n whose associated binding is of the form $n{:}e$ cannot be reduced until e becomes a value.

3.2.3 Environment Evaluation

New environments are added to the global state via the **map** form. The evaluation of a **map** creates a new environment address. Given a reducible expression[4]

$$\begin{array}{ll} e = [\rho_j \downarrow (\textbf{map} & \\ & id_1 : exp_1 \\ & \ldots \\ & id_k : exp_k \\ & \ldots \\ & exp_{k+1} \\ & \ldots \\ & exp_n), ?] \end{array}$$

then

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ e\ \ldots]\ldots\ \rho_n\}}{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ \rho_{n+1}\ \ldots]\ldots\ \rho_n\ \rho_{n+1}\}}$$

where ρ_{n+1} is a new environment address such that $P(\rho_{n+1}) = \rho_j$ and $Contents(M(\rho_{n+1})) =$

$$\begin{array}{l} (\textbf{map} \\ \quad id_1 : [\rho_{n+1} \downarrow exp_1, ?] \\ \quad \ldots \\ \quad id_k : [\rho_{n+1} \downarrow exp_k, ?] \\ \quad \ldots \\ \quad [\rho_{n+1} \downarrow exp_{k+1}, ?] \\ \quad \ldots \\ \quad [\rho_{n+1} \downarrow exp_n, ?]) \end{array}$$

Our semantics allow elements of an environment to be accessed once it has been recorded as part of the global state. All exp_i in the map expression associated with this environment will evaluate in the newly created environment, ρ_{n+1}.

[4]Bindings can appear anywhere inside a **map**. However, for clarity of presentation, we adopt the convention that binding forms appear before any other expression inside a **map**.

The absence of any ordering rule governing the evaluation of expressions in a **map** means that parallel evaluation of a map's subexpressions is allowed. All exp_i in the map expression associated with this environment will evaluate in the environment in which the **map** was created.

3.2.4 Region Selection

The **mlast** operator returns the last element of a map. If $e = [\rho_j \downarrow (\textbf{mlast}\ e_1), ?]$ and e_1 is not an address, then

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ e \ldots]\ldots\ \rho_n\}}{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ [(\textbf{mlast}\ [\rho_j \downarrow\ e_1, ?]), ?] \ldots] \ldots\ \rho_n\}}$$

If $e = [(\textbf{mlast}\ \rho_q), ?]$ where ρ_q is an environment address such that

$$Contents(M(\rho_q)) = (\textbf{map}\ e_1\ e_2\ \ldots\ e_k\ v)$$

or

$$Contents(M(\rho_q)) = (\textbf{map}\ e_1\ e_2\ \ldots\ e_k\ id\ :\ v)$$

where v is a non-$\star$ value, then

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_q \ldots\ \rho_k[\ldots\ e\ \ldots]\ldots\ \rho_n\}}{\{\rho_1\ \rho_2\ \ldots\ \rho_q \ldots\ \rho_k[\ldots\ v\ \ldots]\ldots\ \rho_n\}}$$

Note that the non-strict semantics of maps allows an **mlast** expression to be reduced even if some of the argument map's elements are still under evaluation; the only constraint on the reduction of an **mlast** expression is that the last element in the argument map be a non-$\star$ value.

3.2.5 Specifying the Evaluation Environment

Given a scope-expression: $e = [\rho_j \downarrow\ e_1.e_2, ?]$ in which e_1 is *not* an address, the following rewrite-rule is applicable:

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ e \ldots]\ldots\ \rho_n\}}{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ [[\rho_j \downarrow e_1, ?].[\rho_j \downarrow e_2], ?] \ldots] \ldots\ \rho_n\}}$$

If $e = [\rho_q.[\rho_j \downarrow\ e_2], ?]$ where ρ_q is an environment address, then

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_q\ \ldots\ \rho_k[\ldots\ e\ \ldots]\ldots \rho_n\}}{\{\rho_1\ \rho_2\ \ldots\ \rho_q\ \ldots\ \rho_k[\ldots\ [\rho_{n+1} \downarrow e_2, ?] \ldots] \ldots \rho_n\ \rho_{n+1}\}}$$

where $M(\rho_{n+1}) = M(\rho_q)$ and $P(\rho_{n+1}) = \rho_j$.

In words, the target expression of the scope-expression, e_2, is evaluated in a new environment in which the address of the local map structure, $M(\rho_{n+1})$, is $M(\rho_q)$ and the parent link, $P(\rho_{n+1})$, is ρ_j. Because $M(\rho_{n+1}) = M(\rho_q)$, *i.e.*, because a map structure is shared between two environments, any changes made by e_2 to elements found in the scope-expression's source map are visible to any other expression that happens to access that map.

3.2.6 Sequential Evaluation of Maps

To simplify the presentation, we only consider seqmap expressions that do not define name bindings[5].

If $e = [\rho_j \downarrow (\textbf{seqmap}\ e_1\ e_2\ \ldots\ e_n), ?]$

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ e \ldots]\ldots\ \rho_n\}}{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ [(\textbf{seqmap}\ [\rho_j \downarrow e_1, ?]\ [\rho_j \downarrow\ e_2]\ \ldots\ [\rho_j \downarrow\ e_n]), ?]\ldots]\ldots\ \rho_n\}}$$

Note that we propagate the evaluation environment of the **seqmap** across all sub-expressions of the **seqmap**. This propagation operation effectively builds a *closure* for each of the sub-expressions. If we didn't record the address in this manner, the interpreter would have no history of the expression's proper evaluation environment.

The evaluation of a seqmap's component elements is specified as follows. If

$$e = [(\textbf{seqmap}\ v_1\ [\rho\ \downarrow\ e_2]\ \ldots\ e_n), ?]$$

where v_1 is a non-$\star$ value then

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ e \ldots]\ldots\ \rho_n\}}{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ [(\textbf{seqmap}\ v_1\ [\rho \downarrow\ e_2, ?]\ \ldots\ e_n), ?]\ldots]\ldots\ \rho_n\}}$$

Notice that once e_1 has evaluated, the next expression is tagged as eligible for evaluation; ρ is the expression's evaluation environment. Similar rules apply for each e_i. Once the first $n-1$ component expressions have evaluated, we can transform the seqmap into a map value. Thus, if

$$e = [(\textbf{seqmap}\ v_1\ v_2\ v_3\ \ldots\ [\rho\ \downarrow\ e_n]), ?]$$

[5]Introducing name bindings within a **seqmap** complicates the semantics considerably. As currently defined, the name-lookup rule operates only on maps, not seqmaps. If name-bindings were allowed in a seqmap expression and the binding-value of these bindings could be accessed by other expressions in the same seqmap, the name-lookup rule would have to be modified to support traversing seqmap as well as map objects.

and each of the v_i ($1 \leq i \leq n-1$) are non-$\star$ values then

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ e\ \ldots]\ldots\ \rho_n\}}{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ \rho_{n+1}\ \ldots]\ldots\ \rho_n\ \rho_{n+1}\}}$$

where ρ_{n+1} is a new environment address defined such that $P(\rho_{n+1}) = \rho$ and

$$Contents(M(\rho_{n+1})) = (\textbf{map}\ v_1\ v_2\ v_3\ \ldots\ [\rho \downarrow e_n, ?]).$$

Note that this semantics prevents selection of elements from a **seqmap** until the **seqmap** turns into a **map**; it is possible to support selection of **seqmap** elements, but this would require changing the selection rules slightly.

3.2.7 Environment Abstraction and Application

A **kappa** expression evaluates into a closure-like object whose evaluation environment is fixed to be the kappa's define-time environment. If

$$e = [\rho_l \downarrow (\ \textbf{kappa}\ (id_1\ id_2\ \ldots\ id_n)\ e_1\ e_2\ \ldots\ e_n), ?]$$

then

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ e\ldots]\ldots\ \rho_n\}}{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ e'\ldots]\ldots\ \rho_n\}}$$

where $e' =$

$$[\rho_l \downarrow (\ \textbf{kappa}\ (id_1\ id_2\ \ldots\ id_n)\ (\ \textbf{map}\ e_1\ e_2\ \ldots\ e_n))]$$

We call an evaluated **kappa** expression a **kappa** *closure*. The effect of evaluating a **kappa** object is to collect the expressions constituting its body into a map expression; the evaluation environment of the closure is the evaluation environment of the **kappa** expression. Even though the map expression is unevaluated, the closure itself is regarded as a value. The only operation that can be performed on a closure object is application.

Kappa application in Symmetric Lisp is defined in terms of map evaluation. If

$$e = [\rho_j \downarrow (e_1\ a_1\ a_2\ \ldots\ a_n), ?]$$

then

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ e\ldots]\ldots\ \rho_n\}}{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ [([\rho_j \downarrow e_1, ?]\ [\rho_j \downarrow a_1, ?]\ [\rho_j \downarrow a_2, ?]\ldots\ [\rho_j \downarrow a_n, ?]), ?]\ldots]\ldots\ \rho_n\}}$$

Given an expression, e_{apply} of the form

$$
\begin{array}{l}
[(\ [\rho_l \downarrow (\ \textbf{kappa}\ (id_1\ id_2\ \ldots\ id_n) \\
\qquad\qquad (\ \textbf{map}\ e_1\ e_2\ \ldots\ e_n))] \\
\quad h_1\ h_2\ \ldots\ h_n),?]
\end{array}
$$

where the h_i are either expressions marked for evaluation or values, then

$$\frac{\{\rho_1\ \rho_2\ \cdots\ \rho_k[\cdots\ e_{apply}\ \cdots]\cdots\ \rho_n\}}{\{\rho_1\ \rho_2\ \cdots\ \rho_k[\cdots\ \rho_{n+1}\ \cdots]\cdots\ \rho_n\ \rho_{n+1}\}}$$

ρ_{n+1} is a new environment address defined such that $P(\rho_{n+1}) = \rho_l$ and $Contents(M(\rho_{n+1})) =$

$$
\begin{array}{l}
(\ \textbf{map} \\
\quad id_1\ :\ h_1 \\
\quad id_2\ :\ h_2 \\
\quad \ldots \\
\quad id_n\ :\ h_n \\
\quad e'_1 \\
\quad e'_2 \\
\quad \ldots \\
\quad e'_n)
\end{array}
$$

If e_i is not a name-binding form, then $e'_i = [\rho_{n+1} \downarrow\ e_i, ?]$; if e_i is a name-binding form: $id : exp$, then $e'_i = id : [\rho_{n+1} \downarrow\ exp, ?]$.

The value of the formals defined in the closure object, id_1, id_2, ... id_n are bound to the values of the corresponding actuals found in the application; the evaluation environment of expressions in the **kappa** application-map is the evaluation environment defining the binding-values for the formals and whose parent link is the lexical environment of the **kappa**. The map expression that contained the elements constituting the **kappa**'s body is removed in the map returned by the application; its evaluation environment is used to determine the parent link of this application.

The application semantics of **kappa** is akin to a "parallel call-by-value"[6] semantics – the body of the kappa can be evaluated concurrently with the evaluation of the arguments since both the reduction of the actuals and the map instantiation of the kappa template can be performed simultaneously; the evaluation of the kappa-closure also proceeds concurrently with evaluation of the actuals. The reader should keep in mind that a non-strict semantics does not imply laziness – all actuals in a kappa application are still evaluated; if the evaluation of *body* does not need the value of an actual, however, it does not need to wait for that value to be computed.

[6]The definition of call-by-value as used in sequential languages usually requires that the value of the formal be known *before* the function is applied. Call-by-value, in the context of a non-strict language, refers to a procedure-calling protocol in which both the arguments to the procedure as well the body of the procedure are evaluated in parallel. It is different from a call-by-need or lazy evaluation semantics in that actuals are evaluated independent of whether they are actually required in the procedure.

A lambda expression evaluates in the same way that a kappa does: a lambda evaluates into a lambda-closure with the same structure as a kappa's. Suppose that e_{apply} is as defined above except that it is a lambda-closure (rather than a kappa closure). To reduce e_{apply}, we use the following rewrite-rule:

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ e_{apply}\ldots]\ldots\ \rho_n\}}{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ [(\mathtt{mlast}\ \rho_{n+1}),?]\ldots]\ldots\ \rho_n\}}$$

In other words, the result of applying a lambda-closure is the value of the lambda body in the template structure that would have been generated by the equivalent kappa expression. Note that this definition assumes that the lambda body always appears last in the generated lambda closure; the choice of making the body the last element in the closure is arbitrary and may actually be considered an "asymmetry" in the language. In the abstract, the body may appear anywhere within the lambda closure so long as the appropriate selector operation is used at the time of application.

3.2.8 The Fill Operation and General Assignment

There are several possible ways a `fill` expression may be reduced; the one actually chosen is dependent on the structure of the expression itself (*e.g.,* whether an identifier or an index is given as the `fill`'s arguments) as well as the contents of the region to be filled.

If the region to be filled-in is not empty, the `fill` expression is reducible *only if* the value to be dropped into the region is *equal* to the value currently resident in the region; it is not possible to fill a non-empty region with a value distinguishable from the item currently found in that region.

We can state our definition of equality precisely as follows:

Definition 3.4 *Two values v_1 and v_2 are equal (written $v_1 = v_2$) iff*

- *v_1 and v_2 are primitive constants,* i.e., *integers, booleans, and strings and are equal under the obviouss definition of equality for these types.*
- *If v_1 is a* **kappa***-closure of the form:*

  $[\rho_l \downarrow (\ \mathbf{kappa}\ (id_1\ id_2\ \ldots\ id_n)\ (\ \mathbf{map}\ e_1\ e_2\ \ldots\ e_n))]$

  then v_2 is a **kappa***-closure of the form:*

  $[\rho_l \downarrow (\ \mathbf{kappa}\ (id_1\ id_2\ \ldots\ id_n)\ (\ \mathbf{map}\ e_1\ e_2\ \ldots\ e_n))]$

  A similar definition applies for `lambda`*-closures.*

- *v_1 and v_2 are map values such that*

  1. *$Size(v_1) = Size(v_2)$*
  2. *$v_1[i] = v_2[i]$ for all $i, 1 \leq i \leq Size(v_1)$*
  3. *If the i^{th} region in v_1 is named x, then the i^{th} region in v_2 is named x and vice versa.*

- *v_1 and v_2 are environment addresses that reference map values such that*

$$Contents(M(P^i(v_1))) = Contents(M(P^i(v_2)))$$

for all $i \geq 0$.

There is a single rewrite associated with each of the different forms fill may have.

Case 1: Let $e = [\rho_j \downarrow (\texttt{fill}\ x\ e_1), ?]$. Suppose that there exists an $e' = x : v'$ such that e' occurs in a map whose address is $M(\rho_h)$ where $\rho_h = P^m(\rho_j), m \geq 0$. Suppose, furthermore, that there does not exist $f < m$ such that $M(P^f(\rho_j))$ is the address of a map that defines a region named x. Then

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ e \ldots]\ldots\ \rho_n\}}{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ [(\texttt{fill}\ \rho_h.x\ [\rho_j \downarrow\ e_1, ?]), ?]\ldots]\ldots\ \rho_n\}}$$

Case 1a: Let $e = [(\texttt{fill}\ \rho_h.x\ v), ?]$ where ρ_h is as defined in *Case 1*. Then, for $v' = \star$ or $v' = v$,

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_h[\overbrace{\ldots\ x : v'}^{r}\ldots]\ldots\ \rho_k[\ldots\ e \ldots]\ldots\ \rho_n\}}{\{\rho_1\ \rho_2\ \ldots\ \rho_h[\overbrace{\ldots\ x : v}^{r}\ldots]\ldots\ \rho_k[\ldots\ v \ldots]\ldots\ \rho_n\}} \qquad (*)$$

Case ^{a}: Let $e = [\rho_j \downarrow (\texttt{fill}\ e_1.x\ e_2), ?]$. Then

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ e \ldots]\ldots\ \rho_n\}}{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ [(\texttt{fill}\ [\rho_j \downarrow\ e_1, ?].x\ [\rho_j \downarrow\ e_2, ?]), ?]\ldots]\ldots\ \rho_n\}}$$

Case 3: If $e = [\rho_j \downarrow (\texttt{fill}\ e_1[e_2]\ e_3), ?]$, then

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ e \ldots]\ldots\ \rho_n\}}{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ [(\texttt{fill}\ [\rho_j \downarrow e_1, ?].[\rho_j \downarrow\ e_2, ?]\ [\rho_j \downarrow\ e_3, ?]), ?]\ldots]\ldots\ \rho_n\}}$$

Given an expression $e = [(\texttt{fill}\ \rho_h[r]\ v), ?]$, then for $v' = vor\star$, e can be reduced by applying the following rule:

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_h[\overbrace{\ldots\ v'}^{r}\ldots]\ldots\ \rho_k[\ldots\ e \ldots]\ldots\ \rho_n\}}{\{\rho_1\ \rho_2\ \ldots\ \rho_h[\overbrace{\ldots\ v}^{r}\ldots]\ldots\ \rho_k[\ldots\ v \ldots]\ldots\ \rho_n\}}$$

Suppose, without loss of generality, that the r^{th} region in ρ_h is of the form: $x : v'$ where v' is as defined above; is then reduced using rule (*).

Symmetric Lisp contains an assignment operator, **set**, that is used to redefine name bindings. The **set** operator is a generalization of the **fill** operator in that it allows any region, regardless of whether it is empty or not to be mutated. Thus, unlike **fill** which performs an equality check between the value that is to fill a region and the contents of the region itself if the region is not empty, **set** replaces the contents of the region without performing such a check. Notwithstanding this difference, the rewrite rule for **set** is identical to that of **fill** and is omitted here. Readers should note that both **fill** and **set** are strict in both arguments – the value replacing the contents of a region must be known before any substitution takes place.

3.2.9 Conditional Expressions

Conditional expressions are written using the **if** primitive; **cond** is to be thought of as a macro built using **if**.

If $e = [\rho_j \downarrow (\mathbf{if}\ p\ e_t\ e_f), ?]$ then

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ e \ldots]\ldots\ \rho_n\}}{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ [(\mathbf{if}\ [\rho_j \downarrow\ p, ?]\ [\rho_j \downarrow e_t]\ [\rho_j \downarrow\ e_f]), ?]\ldots]\ldots\ \rho_n\}}$$

In order to remember the proper evaluation environment that the arms are to evaluate within, we propagate the evaluation environment into both arms without marking them (yet) for evaluation.

If $e = [(\mathbf{if}\ \mathbf{t}\ [\rho_j\ \downarrow\ e_t]\ [\rho_j\ \downarrow\ e_f]), ?]$ then

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ e \ldots]\ldots\ \rho_n\}}{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ [\rho_j \downarrow\ e_t, ?]\ldots]\ldots\ \rho_n\}}$$

If $e = [(\mathbf{if}\ \mathbf{nil}\ [\rho_j\ \downarrow\ e_t]\ [\rho_j\ \downarrow\ e_f]), ?]$ then

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ e \ldots]\ldots\ \rho_n\}}{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\ldots\ [\rho_j \downarrow\ e_f, ?]\ldots]\ldots\ \rho_n\}}$$

Once the value of the predicate is known, the appropriate arm can be tagged for evaluation.

$$\begin{tikzcd}
 & M \arrow[dl] \arrow[dr] & \\
M1 \arrow[dr, dashed] & & M2 \arrow[dl, dashed] \\
 & M3 &
\end{tikzcd}$$

Figure 3 : The Diamond Property

3.3 Confluence

A rewrite-rule system in which the final state produced from some initial state is invariant under the reduction sequence taken by the interpreter implies that the language described by the rewrite-system is *determinate.* An important question that may be asked given the rewrite-rules for Symmetric Lisp is whether the rewrite-rules are indeed *confluent*: is the order in which reductions are taken relevant in determining the value of a program?

To make our discussion precise, we define the notion of confluence formally as follows:

Definition 3.5 *Let a* state transition relation, $\longrightarrow_R$, *be a binary relation:*

$$\longrightarrow_R \subseteq (State \times State)$$

Then, $\longrightarrow_R$ *satisfies the* diamond property *if*

$$\forall \text{ states } M, M_1, M_2\ [M \longrightarrow_R M_1 \wedge M \longrightarrow_R M_2] \Rightarrow \exists M_3 \text{ s.t. } M_1 \longrightarrow_R M_3 \wedge M_2 \longrightarrow_R M_3$$

We say that a state transition relation, $\longrightarrow_R$ *, satisfies the* Church-Rosser *property if:* $\Longrightarrow_R$, *the reflexive, transitive closure of* $\longrightarrow_R$ *satisfies the diamond property.*

The reason why $\longrightarrow_R$ is said to satisfy the "diamond" property can be best understood if we consider the definition statement graphically. (see Figure 3).

The diamond property captures the notion of confluence: informally, it states that any two reductions of a term themselves ultimately reduce to a common term. Thus, regardless of which reductions one takes, the final answer is guaranteed to be the same.

To help us think about this problem, it is useful to view Symmetric Lisp in terms of a three-tier stratified hierarchy of languages: at the lowest level, we consider a subset of the language (call it SL_0) containing arithmetic and environment-related operators, abstraction, conditionals and constants. It is easy to convince oneself that such a subset is indeed confluent; indeed, it is straightforward to exhibit a transformation that takes Symmetric Lisp expressions in this subset to terms in the first-order lambda-calculus which is well-known to have a confluent rewrite-system. Map expressions, for example, can be transformed into functions that take names to either a ground value or an error if the name is not defined by the map. A scope-expression could be understood as a function that applies its first argument (which, in this example, would be a map function) to each free name in its second argument; if the application yields an error, then the name is applied to the map function defining the evaluation environment of the scope-expression. Transformations of the other applicative operators can be performed along the same lines. A purely applicative subset of Symmetric language, in other words, is determinate.

The topmost-tier in the hierarchy consists of the full language including general assignment. Because of the language's non-strict operational semantics, one can exhibit simple programs that are non-determinate: because **set** destructively mutates a region, multiple **set**'s on the same region can lead to non-determinism: the order in which the assignments occur affects the final value of the region. General assignment in the presence of non-strictness leads to non-determinism.

The most interesting level in this hierarchy is the middle one. This level (call it $SL_0 + F$) consists of the applicative subset defined by level one augmented with **fill** and empty regions, but does *not include* general assignment. Are the rewrite-rules which define terms constructible from this level also confluent?

To relate confluence to our restricted subset, we need to define the rewrite-rules in terms of a state transition relation:

Definition 3.6 *Let $\mathcal{I} =< SL_0+F, \longrightarrow_{\mathcal{I}}>$ be an algebraic structure (called an* Abstract Reduction System*) consisting of a language $SL_0 + F$ and a relation $\longrightarrow_{\mathcal{I}}$ on states defined as follows:*

If

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_k[\overbrace{\ldots\ e}^{r}\ldots]\ldots\ \rho_m\}}{\{\rho'_1\ \rho'_2\ \ldots\ \rho'_k[\overbrace{\ldots\ e'}^{r}\ldots]\ldots\ \rho'_n\}}$$

(where $m \leq n$ and ρ'_i is identical to state ρ_i except for any substitutions to terms in ρ_i specified by the reduction of e and where the reduction of e yields e')

then

$$\{\rho_1\ \rho_2\ \ldots\ \rho_k\ \ldots\ \rho_n\} \longrightarrow_{\mathcal{I}} \{\rho'_1\ \rho'_2\ \ldots\ \rho'_k\ \ldots\ \rho'_n\}[\rho'_k[r], e']$$

where $S[\rho_k[r], e']$ denotes the state derived by substituting e' for the r^{th} region in environment ρ_k found in S.

We posit the following conjecture:

Conjecture 3.1 $\longrightarrow_{\mathcal{I}}$ *satisfies the diamond property.*

This statement asserts that the order in which reducible expressions are reduced will not affect the meaning of the program: if e_1 and e_2 are two reducible expressions in some state S such that the reduction of e_1 produces a new state S_1 and the reduction of e_2 produces a new state S_2, then there is another state S_3 that can be reached from both S_1 and S_2 by the reduction of expressions in those states resp.

The reason why this should be the case is intuitively obvious: a `fill` operation can either fill an empty region (which, according to the operational semantics, cannot be read or copied by any expression) or fill a value *identical* to the one currently occupying the region being filled. In this case, the `fill` operation effectively behaves as a "no-op" instruction: its execution has no perceptible *non-local* effect on the global state.

What is the effect on the global state if two `fill` operations try to fill different values for the same region? Based on the operational semantics, the second `fill` operator will never get reduced since the reduction of a `fill` expression is predicated on the fact that either the region being filled-in is empty or contains a value identical to the value of `fill`'s second argument. A race-condition exists here that must be taken into consideration when we try to formally prove the confluence of the $SL_0 + F$. Consider the following program:

```
(map
   x : *
   (fill x 1)
   (fill x 2))
```

Depending upon which `fill` is reduced first, the value of `x` will be either 1 or 2. Moreover, there is no rewrite-rule that indicates that such a program is actually in error. We, therefore, introduce the following extra "blowup" rule to handle such a case:

Case 4: Let e be $[(\mathtt{fill}\ \rho_j.x\ v), ?]$. Suppose that there exists an $e' = x : v'$ such that e' occurs in a map whose address is $M(\rho_h)$ where $\rho_h = P^m(\rho_j), m \geq 0$. Suppose, furthermore, that there does not exist $f < m$ such that $M(P^f(\rho_j))$ is the address of a map that defines a region named *id*. Then, if $v' \neq \star$ or $v' \neq v$ then

$$\frac{\{\rho_1\ \rho_2\ \ldots\ \rho_h[\overbrace{\ldots\ x : v' \ldots}^{r}]\ldots\ \rho_k[\ldots\ e \ldots]\ldots\ \rho_n\}}{\{\top\}}$$

(Similar rules apply to the other forms of **fill**.) $\top$ defines the error state: the value of a program containing multiple **fill** operations on the same region is in error. This constraint is severe, but is necessary to satisfy the confluence property. To support this rule, we must define a state to be either a finite set of environments or the special error value $\top$. The value of a program that reduces to a state bound to $\top$ is $\top$.

Note that an approach in which errors are propagated locally would not be sufficient. In other words, a semantics that replaced the contents of the region named x with a special error value, (say, **error**), must still address the fact that other expressions may have read x while it was still bound to v'; only expressions that subsequently read x will see the **error** flag. Such a program is clearly non-determinate since the value of x seen by an expression depends upon the order in which the reductions are performed - expressions reduced before the **error** flag is stored will see v'; expressions reduced afterwards will see **error**. By simply reducing the entire state to an error value, we can guarantee that a program containing two **fill** operations on the same region will still yield the same value (here a state defining $\top$) *even* if the values being filled are different.

3.4 Summary

The formal operational semantics presented in this chapter could be used as the basis for the implementation of a Symmetric Lisp interpreter; since the rewrite-rules are unambiguous, the interpreter can determine which reduction to perform by simply examining the expression and the necessary elements of the global state. The rules themselves are more cumbersome than the rewrite rules for standard combinatory systems like the λ-calculus; the added complexity is clearly due to the presence of an abstract store and the need to distinguish between addresses and primitive values. However, the fact that maps are used to represent both programs and data avoided the need to have a separate domain of data structures and program constructs; the definition of normal form was sufficient to define the notion of data structure.

The remainder of the thesis is devoted to describing applications and implementation of Symmetric Lisp. Although we will be informal in our description, the precise meaning of all the examples given in the thesis can be understood using the formal semantics given in this chapter. The reader should feel free to use the semantics presented here as a formal reference manual for the language.

Chapter 4

Paradigms and Applications

Central to the symmetric programming model is the notion that program structures are data objects; a program has a well-defined "shape" and it can be examined and built in the same way that a conventional data structure can. What are the merits of such a model and what paradigms does it support well? This chapter investigates these questions by examining three closely-related issues: section 4.1 examines the implications of *uniformity* of program and data for program design and methodology, section 4.2 discusses the meaning of *modularity* in the symmetric language context focusing on how first-class naming environments can be used to build modular programs, and section 4.3 examines the synergistic interaction between first-class environments and *parallelism.*

4.1 Uniformity

What are the ramifications for program-building if program structures are treated identically to data structures? Most programming models preserve a clear distinction between the role of a data structure and that of a program. In Symmetric Lisp, such a separation does not exist: because any program can be thought of and used as a data structure, the Symmetric Lisp programmer has great flexibility to express a wide range of different program constructs found in other languages.

When it is said that one can model program structure P found in language L in Symmetric Lisp, it does not mean that the Symmetric Lisp version is identical to the original structure, but that it captures the essence of the original. It may not be the case that a Symmetric Lisp programmer will choose to write Symmetric Lisp programs in the style encouraged by another language, but it is important that the abstractions provided by different program constructs found in other languages be expressible simply (and efficiently) using maps so that in cases where such abstractions are indeed useful, the programmer has the capability to implement them without great effort.

To begin with, consider the following (superficially different) program structures found in several modern programming languages:

- An Algol-60[52] block or a Scheme[31, 59, 58] **letrec** statement used to define a local naming environment. A block statement or a let expression defines a collection of local names and evaluates some expressions using the bindings given to those names.
- An Ada[3] package or a Modula[71] module used to encapsulate a related collection of data and procedures. These structures serve as the basic program structuring mechanism in these languages. It is possible to instantiate many copies of a package or module with each instantiation having its own copy of local data.
- A CSP-style[42] "cobegin" or multi-tasking structure of the form:

  cobegin *<local variables> S1; S2; ... Sn* **coend**

  used to create *n* concurrent threads of execution within a local namespace. The spawned processes communicate either through shared data or through explicit communication primitives (*e.g.*, the message passing operators found in CSP).
- A guarded-command[28] construct found in languages such as Concurrent-Prolog[55]. A guarded command is a program structure that consists of a series of statements each of which is prefixed by some boolean expressions (called *guards*). A statement is eligible for execution only when all of its associated boolean expressions are true. If several guards in different statements become satisfied simultaneously, the statement finally chosen to be executed is chosen arbitrarily.
- Structures used to build infinite data objects such as streams found in languages like Miranda[65] or VimVal[26]. In languages that are not based on a demand-driven evaluation model, special program structures are needed to delay the evaluation of expressions. These structures typically consist of a reference to the form whose evaluation was deferred and a reference to the binding environment in which the expression is to be evaluated[1].
- The *class* construct found in inheritance-based object-oriented languages such as Simula[23] or Smalltalk[38] is a structuring tool that allows the programmer to encapsulate and share information among objects of many different types. Objects can inherit operations defined

[1] **Suspensions** first proposed in [33] implement a lazy cons cell in this way. (An Algol-60 thunk can be thought of as basically a suspension; it differs from suspensions in that it was used only by the implementation and there only to realize call-by-name parameter passing; it was not available to the programmer for building lazy data structures.) The **delay** special form found in MIT Scheme and in MultiLisp is a generalization of a suspension that can be used to delay any expression.

on their supertypes; when an object receives a message[2], it first searches the methods[3] defined in its class to see if the appropriate procedure is defined there; if it is not, search proceeds to the object's superclass and so on until a matching method is found.

What do these constructs have in common? On the surface, not much; some deal with parallelism, others with namespace management, and still others with controlling expression evaluation. Interestingly enough, all these structures have a straightforward interpretation in Symmetric Lisp. The parallel evaluation semantics of maps makes them suitable for describing parallel constructs like cobegin-coend statements; the fact that maps define first-class *structure-based* naming environments[4] means that they can be used to model a variety of different namespace manipulating constructs; the implicit block-on-uncomputed-values rule and the fact that empty regions can be used to explicitly synchronize the evaluation of expressions give the Symmetric Lisp programmer flexibility in controlling expression evaluation. It is not the case that a map structure is fortuitously useful in several unrelated domains: we argue that the semantics of maps faithfully captures, and in many cases extends, the semantics of these other program constructs.

A comparison between Symmetric Lisp's approach to implementing these constructs and their realization in other languages is given below.

4.1.1 Blocks and Local Naming Environments

A structure like the Algol block

```
begin integer x, y, z;
      x := expr1; y := expr2; z := expr3;
      expr4
end
```

can be thought of as the following map:

```
(map
   priv x : expr1
   priv y : expr2
   priv z : expr3
```

[2]A message is a request for an object to carry out some operation.

[3]A method is a procedure defined on an object that describes how the object should carry out a particular operation.

[4]*Structure-based* as opposed to *Function-based*: a naming environment could be thought of as a function from names to values, but it is more appropriate in the Symmetric Lisp context to view a naming environment as a record in which each record field defines a binding in the environment.

```
    expr4)
```

Unlike an Algol block, the above map yields a value; the value it yields is an environment (defines names) but, because it has a well-defined spatial interpretation, it can be treated like a data structure. Note that because the names defined in the above expression are declared as private, they are not visible to expressions outside of the map – this is consistent with the scope rule for local names found in an Algol block.

Since the map version of the block behaves like a data structure, its elements can be selected by position as well as by name. To turn the above map into a Scheme-style letrec,

```
(letrec ((x expr1)
         (y expr2)
         (z expr3))
  expr4)
```

we need only wrap a selector operation around the map to retrieve the value of **expr4**. Thus, the expression:

```
(mlast
  (map
     priv x : expr1
     priv y : expr2
     priv z : expr3
     expr4))
```

returns the value of **expr4** in the block. The expressions local to the block themselves evaluate in the same environment; thus, expressions may be mutually recursive. This map structure is a ·ually, therefore, a bit more general than the Scheme letrec. In Scheme, it must be possible to evaluate each of the binding expressions without having to refer to the value of any locally-defined variable. Thus, while a Scheme letrec can define mutually-recursive procedures (since name references in a procedure body do not refer to a variable's value at procedure definition time), it cannot support more general mutual recursion. The concurrent evaluation semantics of maps and its block-on-uncomputed-values rule makes its evaluation behaviour more similar to letrec constructs found in lenient[53][5] or lazy[65] languages.

4.1.2 Packages and Libraries

The package is the main structuring and encapsulation unit in Ada (as is the module in Modula). The role of the package as an encapsulation device is handled in Symmetric Lisp using maps

[5] A lenient language is non-strict but not lazy.

and kappas. Consider the following Ada package declaration;

```
package stack is           --- the specification part
   procedure push(x:real);
   function  pop return real;
end;
package body stack is      --- the implementation part
   max : constant := 100;
   s : array(1..max) of real;
   ptr: integer range 0 .. max;
   procedure push (x:real) is
     <  code for push >
   end push;
   function pop return real is
     <  code for pop >
   end pop
   begin
     ptr := 0
   end stack
```

The package specification describes the interface between the package body and its users. In the example, only the two procedures, **push** and **pop** are visible outside the package; the stack representation, **s**, and the other local variables in the stack are hidden from outside expressions. The body of the package is executed when the package is declared and is used to initialize local variables defined in the package.

The items declared in the specification of a package may be used by expressions outside the package using dot notation:

```
stack.push(100);
   :
x := stack.pop();
```

The above package could be implemented in Symmetric Lisp by the following map:

```
stack : (map
          priv max : 100
          priv s : (map)
          priv ptr : 0
          push : (lambda (x)
                    <  code for push >)
          pop  : (lambda (y)
                    <  code for pop >))
```

Here again, the use of the priv binding in the map restricts the visibility of the names it prefixes. The above map represents an instance of an abstract data type; the representation type of the abstraction is **s** and the operations defined by the abstraction are **push** and **pop**. Users of the stack do not have access to its representation; users can only manipulate the representation via procedures **push** and **pop**. To access these procedures, one uses notation similar to the one used in Ada:

```
(stack.push 100)
   ⋮
x : (stack.pop)
```

Because functions are lexically-scoped, they have access to the private names defined in the stack even though expressions found outside do not.

The **generic** package construct allows one to build package templates. In the stack example given above, for example, the bounds of the stack is a manifest constant; to parameterize this component, one would express the stack as follows:

```
generic
   max : integer;
package stack is
   <  same specification as above >
end;
package body stack is
  s : array (1 .. max) of integer;
        ⋮
end stack;
```

To instantiate a stack that is to hold 100 elements, one now writes:

```
package my-stack is new stack(100)
```

Note that to build a generic package template not only involves defining a new package declaration form that allows the programmer to specify which items in the package are to be parameterized, but also requires a new notation for template instantiation. The basic reason for this lies in the distinction made between the package program structure and the other data and program structures found in the language; one cannot combine a package structure with other available program or data constructs.

In Symmetric Lisp, parameterized package-like structures can be written using the kappa construct. The above generic package could be expressed as:

```
stack : (kappa ((priv max))
          priv s : (map)
          priv ptr : 0
          push : (lambda (x)
                    < code for push >)
        pop   : (lambda (y)
                    < code for pop >))
```

To instantiate a new stack of 100 elements, one writes:

```
my-stack : (stack 100)
```

This kappa acts like an abstract type declaration. Creating a new instance of such a type involves instantiating a map template by kappa application. The map returned by the above application would have as its first field a name-expression that binds **max** to 100.

Unlike packages and modules, maps may be nested to any depth; thus, one can build libraries of package-like structures as well. For example, suppose one wishes to collect the specifications of all structure types into a single library called **structures**. Such a library could be written in Symmetric Lisp as follows:

```
structures : (map
                 stack : (kappa ((priv max))
                             < as shown above >)
                 queue : ...
                 bounded-buffer : ...
                 ...)
```

To create a stack of 100 elements, one now evaluates **structures.(stack 100)**.

Note that the **structures** map is used basically as a *data structure* containing a number of named fields; the contents of the fields are kappa objects that have the operational behaviour of a generic package *program structure*. An Ada package or a Modula module, unlike a Symmetric Lisp map, is *not* a first-class structure: it cannot be built into data structures, nor can it be passed as arguments to functions.

4.1.3 Process Instantiation

The last two examples have shown how to use maps to model program structures that behave like local naming environments. In describing the implementation of these constructs in Symmetric Lisp, the map's parallel evaluation semantics was secondary to its function as a naming environment. Note, however, that because elements of a map evaluate concurrently, a map structure defines a locus of parallel activity wherein each of its elements define a separate parallel execution thread. Thus, the CSP process-forking statement:

$S_1 \mid S_2 \mid \ldots \mid S_n$

where the S_i are arbitrary expressions, translates trivially into the map expression

(map S_1 S_2 ... S_n)

A discussion on interprocess communication structures is deferred until Section 4.3.

4.1.4 Non-Deterministic Program Constructs

Guarded clauses are used extensively in several concurrent logic programming languages. In Concurrent Prolog (CP)[55], guards serve as the primary synchronization and control construct. A (flat) guarded clause is a statement of the form

$$A \leftarrow G_1\ G_2 \ldots G_m \mid B_1\ B_2 \ldots B_n$$
$$(m, n \geq 0)$$

The *commit* operator (|) separates the right-hand side of the rule into a *guard* and a *body*. A procedural interpretation of a guarded clause is as follows: the reduction of a process A_1 using the above guarded clause suspends until A_1 is unifiable with A and the guard's component elements all evaluate to true. Each of the G_i are constrained to be simple test predicates, *e.g.*, predicates to test the type of a variable, or simple logical or relational operators. Because a guard prevents a process from continuing execution until all of its component test predicates become satisfied, it is a mechanism for synchronizing process activity in much the same way that condition variables found in monitor-based languages are used to synchronize access to a monitor among concurrently executing processes. A guard is an important program structure for concurrent systems because it can be used to express structured non-determinism and mutual exclusion.

In general, A may be on the left-hand side of a number of guarded commands:

$$A \leftarrow < guard_1 > \mid\ < body_1 >$$
$$A \leftarrow < guard_2 > \mid\ < body_2 >$$
$$\ldots$$
$$A \leftarrow < guard_n > \mid\ < body_n >$$

We can capture the behaviour of a guarded clause system in Symmetric Lisp through use of open-maps (open-maps were defined in Section 2.2). To represent the above system, one defines two maps, one to hold the guards and the other to hold the bodies. The i^{th} element in the body map is a n . lary function whose body is the i^{th} body expression in the guard system. The i^{th} element in the guard map is a nullary procedure whose body is the conjunction of the guards found in the i^{th} guarded command in the guard system.

To execute a guard-system, one defines a function called **guard-eval**:

```
guard-eval : (lambda (G B)
                   M : (open-map.create)
                   (onto (lambda (g i)
                            (if (g)
                                (attach M i)))
                      G (iota 1 (msize G)))
                   (B[(mcar M)]))
```

The function takes two arguments, a guard map, G, containing the value of each of the guard predicates and a body map, B, containing the body expressions associated with each of the guards. It defines one local name, M initially bound to an empty open-map. The **onto** function (described in Section 2.1.9) applies its function argument pointwise to each of its argument maps. The **onto** expression in the above example is equivalent to the following map:

```
(map
   (if (G[1]) (attach M 1))
   (if (G[2]) (attach M 2))
          ...
   (if (G[(msize G)]) (attach M (msize G))))
```

Each of the $G[i]$ applications evaluate to a boolean value. Whenever any of the applications evaluate to true, the index of the guard in the guard map is attached to open-map M. If there is more than one guard clause that evaluates to true in the guard system, the index first attached to M is arbitrarily determined since all the guard tests occur in parallel. The last expression in the **guard-eval** function, **(B[(mcar M])**, applies the i^{th} procedure in B if i is the value of the first element in open-map M. The application can take place only when an attach on the open-map occurs; if all the guards fail, no attach is performed and no body expression is evaluated: the **mcar** application would wait indefinitely for the empty region found in the first element of the open-map to be filled.

How does the Symmetric Lisp solution compare with the CP implementation? Consider how CP uses guard clauses. In Concurrent Prolog, a program is simply a finite set of flat guard clauses. The CP interpreter repeatedly (and non-deterministically) attempts to reduce some goal in the resolvent by trying to unify it with some guard clause. In the case where there is no unifiable clause, *e.g.,* when all guards in the guard-system yield false, the program halts. Note that unlike sequential Prologs, CP doesn't support a backtracking or failure semantics - once a goal and clause are chosen for unification and a process has reduced itself using the bindings produced by the unifier, it cannot later backtrack and acquire new bindings. In the Symmetric

Lisp formulation given above, there is also no failure semantics – once the **guard-eval** function is invoked, the application cannot subsequently be retracted. On the other hand, whereas the CP program would halt if all guards yielded false, the Symmetric Lisp program would suspend forever waiting for a result to be returned by the function.

It is straightforward to extend the example to handle termination. We maintain a counter that is incremented whenever a guard evaluates to false; if during the evaluation of **guard-eval**, the size of the counter equals the number of guards, we know that all guards have failed.

```
guard-eval-with-termination :
   (lambda (G B)
      M : (open-map.create)
      fail-count : 0   fail-lock : lock
      (onto (lambda (g i)
         (if (g)
             (attach M i)
             (holding fail-lock
                (seqmap
                   (inc! fail-count)
                   (if (= fail-count (msize G))
                       (attach M -1))))))  ; some fencepost value
         G (iota 1 (msize G))))
      (if (equal (mcar M) -1)
          nil
          (B[(mcar M)])))
```

The body of the **holding** expression increments the **fail-count** counter and then tests to see whether its value is equal to the number of guards in the guard system. If it is, some fencepost value is attached to **M**. If all guards fail, the evaluation of **(mcar M)** will yield this value and the result of the function is **nil**; if a guard does evaluate to true, then a valid index will be attached to **M** and the corresponding procedure in **B** will be evaluated.

Notice that the Symmetric Lisp solution was derived from a not-so-obvious application of empty regions and open-maps. Empty regions (in connection with open-maps) served as an efficient synchronization mechanism between asynchronously executing processes sharing access to a communication stream; in this example, however, it was used for a completely unrelated purpose: it served to explicitly block evaluation of an expression until a user-specified condition was realized. Because any expression that tries to read an empty region immediately blocks, we can guarantee that the evaluation of a body expression takes place only if the corresponding guard is true.

4.1.5 Lazy Data Structures

Languages based on a normal-order (or *lazy evaluation*) semantics evaluate expressions only when absolutely necessary, *i.e.*, when the expressions' value is needed in order to continue with the computation. Among other things, lazy languages can express infinite data structures such as streams; the elements of such data structures are produced only when a demand is made for them by a consumer of the stream. Stream structures are very useful in expressing many different kinds of producer/consumer relationships as well as in modeling history-sensitive computation. Because of their utility, several non-lazy languages (*e.g.*, MultiLisp and MIT-Scheme) have built-in support for lazy stream structures through the use of explicit **force** and **delay**-like constructs. A delayed object is not evaluated until some expression forces it; in particular, if a stream is implemented as a list, one can effectively build infinite streams by always delaying the cdr of the list. The cdr presumably will be a recursive function call to the stream producer. When forced, the next element is produced, and another delayed object is generated that represents the following element in the list.

One can build infinite data structures in Symmetric Lisp by treating an empty region as a synchronization point between producers and consumers of the data structure. A delayed object is implemented using empty regions and the seqmap form. An empty region is used as a *suspension* – it precedes the expression which is to be delayed. By imposing a serial evaluation order on map expressions, it is possible to use $\star$ as a blocking mechanism that prevents the evaluation of other expressions. To force a delayed expression, one uses the `fill` operator to plug the empty region with a value. Once the hole is plugged, the delayed expression can begin evaluation. If the delayed expression is a function call to a recursive stream producer, then the object yielded by evaluation of the forced expression will be another seqmap of the same form.

To make the discussion more concrete, consider the following Miranda[65] expression:

```
ones = 1 : ones     ; : is the Miranda cons operator
```

This expression defines a stream generator that produces an infinite stream of ones. Because of the lazy evaluation semantics of the language, however, the stream is generated incrementally and only on demand.

The ability to serialize evaluation of map structures and the fact that regions may be used as synchronization points makes it possible to implement lazy evaluation in Symmetric Lisp. For example, the above Miranda expression could be written as follows (ignoring issues of syntactic clarity for the moment):

```
ones : (kappa ()
          first : 1
          suspension : ⋆
          rest : (mlast
                    (seqmap
                       suspension
                       (ones))))
```

A functional-language stream producer that produces its elements on demand is expressed in Symmetric Lisp as a function that returns a map with three regions: (1) a region named **first** that is bound to the first element in the stream, (2) a region called **suspension** initally bound to an empty region, and (3) a region named **rest** that is bound to expression responsible for producing the remaining elements in the stream. To prevent an infinite recursion, **rest** is initially bound to a seqmap whose first element refers to the suspension and whose second element is a recursive call. Because seqmap begins evaluation of its $i+1^{st}$ element only after its i^{th} element has finished evaluating[6], the recursion will not proceed until the empty region to which the suspension is bound is filled-in. In other words, the region bound to **rest** will not get evaluated until a demand for a new element is made. This follows from the fact that the **mlast** operator operates only over maps (not seqmaps) and a seqmap form turns into a map only after all its elements have evaluated. Once such a demand is made, the recursion unfolds once more only to be stopped again as yet another suspension is encountered.

There are two supporting functions on delayed streams:

```
first : (lambda (stream)
           stream.first)
rest  : (lambda (stream)
          (fill stream.suspension t)
          stream.rest)
```

Thus,

```
(first (ones)) ⟹ 1
(rest  (ones)) ⟹ (map
                      first : 1
                      suspension : ⋆
                      rest : (mlast
                               (seqmap
                                  suspension
```

[6]Where by "finished evaluating", we mean "yielded an object".

```
                                        (ones))))
```

Recall that the `fill` operator follows a unification-like semantics. If a rest operation is performed on the same structure twice, `fill` will still succeed (since the region to be filled contains a boolean value which is the same as the value that `fill` is to plug into that region). Because the equality check performed by `fill` is atomic, users of delayed objects don't have to pay the overhead of checking the status of the suspension explicitly. If `fill` didn't obey a unification-style semantics, such a check would be necessary to avoid unnecessarily reproducing the rest of the structure.

Consider another example: The following Miranda function defines an infinite stream of integers beginning from some n:

```
intstreams n = n : intstreams (n+1)
```

One would express this function in Symmetric Lisp as:

```
intstreams : (lambda (n)
                (map
                  first : n
                  suspension : ★
                  rest : (mlast
                           (seqmap
                             suspension
                             (intstreams (+ n 1))))))
```

It's clear that the Symmetric Lisp formulation of the function is more cumbersome than the Miranda one. The extra machinery is due primarily to the fact that we are attempting to support lazy evaluation on top of an eager, "object-oriented" evaluation model. Nonetheless, the verbosity of the Symmetric Lisp solution could be greatly reduced if a simple macro facility similar to that found in Common Lisp were available. Given such a macro facility, it would be easy to define a macro `affix` that would expand to the above map expression:

```
(macro affix (e1 e2)
      `(map
          first : ,e1
          suspension : ★
          rest : (mlast
                    (map
                      (seqmap
                       suspension
                       ,e2)))))
```

Affix would be a name not bound to any region; the compiler would substitute all references to **affix** in the program with the macro body; input expressions would be substituted at the point where they are used in the macro. Thus, the integer stream now could be written as:

```
intstreams : (lambda (n)
                (affix n (intstreams (+ n 1))))
```

This solution is similar to one in which semaphores are used as the synchronization device and P and V like operators regulate access to the end of the structure. In this respect, Symmetric Lisp's solution is similar in style to one that would be written in a language like MultiLisp. A lenient language like Id that uses I-structures for synchronization can't efficiently implement delayed objects because it doesn't have anything similar to the unification-style behaviour of **fill**[7]. One could use a conditional expression to implement delayed objects by having the producer be modelled as a conditional. The predicate of the conditional would be an I-structure and the arms of the conditional would contain the delayed expression. The conditional would evaluate only when a consumer assigns a boolean value to the I-structure that is used as the predicate. Unfortunately, this solution requires an unnecessary conditional test to be executed on every stream demand. In addition, implicitly parallel languages that do not have constructs similar to **seqmap** or I-structures cannot express the explicit synchronization required to delay the evaluation of future elements in the stream.

The Symmetric Lisp solution is deficient in some respects especially when compared with the corresponding implementation in a lazy language such as Miranda. Because a Symmetric Lisp delayed object is represented as an abstraction, built-in operations on ordinary maps (such as onto or composition) cannot operate on maps implementing delayed expressions; separate functions that effectively serve the same role would need to be written specially for delayed structures. Secondly, every demand (even multiple demands on the same element in the structure) would involve a **fill** operation to be executed; although the unification check on **fill** is atomic, it is nonetheless an overhead (similar to the one incurred in the possible Id solution described above) that is avoided entirely in an inherently lazy language. Nonetheless, this example provides a good illustration of how one can exploit the map's program semantic features (*e.g.*, the ability to specify the evaluation order of expressions inside a delayed object using **seqmap**) to build interesting abstractions (infinite streams).

[7]Although there is no reason why such a semantics coul · 1ot be incorporated into the language.

4.1.6 Object-Based Methods and Inheritance

4.1.6.1 Building Objects

Object-based programming refers to a programming methodology in which the basic computational agents in a program are thought of as *objects* that communicate by message-passing. An object retains local state information, *i.e.*, its lifetime may be greater than the expression that created it. An object that receives a message can alter its internal state appropriately based on the content of the message. It is not difficult to support an object-based methodology in any language that allows expressions to have (potentially) infinite lifetimes; any language that supports closures, for example, can also be used to build an object-based system. A Symmetric Lisp map is effectively an object: a map's lifetime may exceed the lifetime of the expression that creates it and the elements of the map constitute its internal state. Message passing may be regarded as expression evaluation within a user-specified environment.

Consider the following object-based implementation of a cons-cell:

```
cons : (kappa (car cdr)
         rplaca : (lambda (new-car) (set car new-car))
         rplacd : (lambda (new-cdr) (set cdr new-cdr)))
```

The definition

```
my-cell : (cons X Y)
```

yields

```
my-cell : (map
            car : X
            cdr : Y
            rplaca : (lambda (new-car) (set car new-car))
            rplacd : (lambda (new-cdr) (set cdr new-cdr)))
```

Evaluating `my-cell.car` yields `X`. This solution is similar in spirit to one that could be formulated in a language like ML[51] that allows record fields to contain procedures[8] except that, unlike a simple record, a map has spatial characteristics – map elements may be selected by position as well as by name; thus, one could also evaluate `my-cell[1]` to yield `X`.

Example: To illustrate this point more clearly, consider the following example: one needs a simulator to calculate the response of a flotilla of ships to a series of threats (*e.g.*, squalls,

[8]Assuming that the record names are recorded in the evaluation environment before evaluation of the lambda expressions begin.

sharks, menacing whales, etc.); assume the threats are fast-moving with respect to the flotilla, so each flotilla member is parked in one place throughout each script.

The flotilla may be represented as a map in which each ship is a named sub-map:

```
flotilla : (map
                grumpy : (map ... )
                sleepy : (map ... )
                    ...
                schwartz : (map ... ))
```

Each ship map includes latitude and longitude fields, which can be updated between runs as appropriate:

```
(set flotilla.schwartz.lat x)
```

Structurally, each ship map will contain a `response` function that can be applied to the description of a threat; the function calculates the appropriate response for this ship. For example, to compute the `schwartz`'s response to threat Q, one evaluates, (`flotilla.schwartz.response` Q). One can determine the response of all ships to threat Q by evaluating:

```
(onto (lambda (ship)
         (ship.response Q))
  flotilla)
```

Because threats are non-uniform (*e.g.*, we consider whales to be more menacing than dolphins), each will be characterized by values along a different set of axes. It is therefore reasonable to model each threat as a map:

```
(map
  behaviour : threat-catalog.whale
  course  : ...
  mass : ...
  velocity : ...
      ...)
```

A threat class is a kappa that returns a new instance of the threat:

```
whale : (kappa (< initialization arguments >)
                behaviour : threat-catalog.whale
                                ... )
```

The `behaviour` of a threat is determined from a threat-catalog that records the characteristics common to all instances of a threat.

```
threat-catalog : (map
                    whale : (lambda (threat-instance)
                                (if (and (threat-instance.hungry ...)
                                       (= threat-instance.course ... )
                                                            ... )
                                        < action >))
                        ... )
```

In responding to a threat Q, a ship evaluates some expression based on the behaviour of Q. Thus, the **response** function in each ship will be structured as:

```
  response : (lambda (threat)
                new-threat : (threat.behaviour threat)
                < respond to new threat>)
```

In responding to a threat, a ship may examine the status of any other ship by referring to it by name; the status of all ships can be determined by iterating over the **flotilla** map:

```
  (onto (lambda (ship)
             < examine ship's status>)
     flotilla)
```

The simulation script could be a map of threat instance applications:

```
  script : (map
              (whale < initialization args>)
                             ... )
```

The simulation function evaluates the **response** function of every ship to every threat:

```
  sim : (lambda (flotilla-map threat-map)
            (onto (lambda (threat)
                      (onto (lambda (ship)
                                (ship.respone threat))
                         flotilla))
               script))
```

Note that, in this formulation, every ship is an object of a fixed number of fields that could be instantiated from a common template (*i.e.*, we could have defined a ship kappa of which a ship would be an instance). A ship could have been expressed as a closure, but then it would no longer be possible to refer to the elements of ship directly by name; one would have to provide a dispatch function to interface between users of the closure and the closure's contents. Languages that provide record types could not represent the flotilla map and threat-catalog map as a record of records because the **sim** and **respond** functions need to iterate over the entire structure; implementing these maps as an array of records would also fail because one also needs to be able to access any element in the flotilla and threat-catalog by name.

Because ships may update their status position and parallel evaluation is the default, programmers need to take care that all responses to one threat are recorded before the next threat is handled. Consider a restructuring of the response function so that it now accepts two arguments, one describing the current state and the other, a new threat; it returns a new state, based on the ship's response to the threat:

```
response : (lambda (curr-state threat)
                new-threat : (threat.behaviour threat)
                < respond to new threat using curr-state>)
```

One can now define a *synchronizing* version of the simulator; this new version is defined in a style similar to programs that may be written in other languages supporting non-strict evaluation[40, 53]; the `sim` function is rewritten as a data structure – an array whose i, j^{th} element will be the response of ship i to threat j. When this data structure is created, each of its components will be an active computation. Some of these computations will depend on others; in particular, the value of the j^{th} entry in row i (the response of ship i to threat j) will depend on the value of the $j - 1^{st}$ (the response of ship i to threat $j - 1$) entry in the same row. The non-strict map-access rules insure that each computation waits until the value it needs is available, and then proceeds. The array of computations quiesces, column by column, into an array of data values. The array can be specified as follows:

```
sim : (generate (i (iota 1 (msize script)))
          (generate (j (iota 1 (msize flotilla)))
             (if (equal j 1)
                 (flotilla[i].response <initial-state> script[j])
                 (flotilla[i].response sim[i,j-1] script[j]))))
```

4.1.6.2 Inheritance

Object-oriented languages supporting inheritance are based on a semantic model in which data is organized into a class hierarchy. Data at any level of the hierarchy *inherit* all the attributes of data higher up in the hierarchy. Every data object is a member of some *class*; a class can be thought of as an object template. Inheritance relations are specified by the way classes relate to one another: objects instantiated from a class A which is a subclass of class B inherit all the procedures (or *methods*) and local variables (or *instance variables*) found in B that are not defined in A. An object, upon receiving a message, dispatches it to the appropriate procedure in its hierarchy which may then proceed to evaluate it. Each object retains local state information that may be used by any of the methods found in its class hierarchy.

There are several competing paradigms for general object-based programming: in Simula-67[23] instances are similar to records with function-valued components and message-passing is realized as field selection and application over these records; Smalltalk[38] (followed by CommonLoops[14] and Flavors[39]) treats message-passing as function call. The first argument in a message is the name of the method to be invoked – an object determines the location of the method by dynamically searching its class hierarchy and applies the method to the remainder of the message. Amber[16, 17] also models objects as records, but expresses inheritance as subtype relations among these records; one can view Amber as a natural object-oriented extension of ML.

Symmetric Lisp views objects as environments, and inheritance as specification of an environment evaluation path. An object will be defined as a map with multi-layer regions; the order of the layers in a region specifies the inheritance hierarchy of the instance variables or methods that the region represents.

In thinking of objects as maps, we see that an instance of a subclass in an inheritance hierarchy is simply an environment that is the "fusion" of the environments defined by the associated instances of all its superclasses. The inheritance hierarchy determines how the environment is to be constructed: nameclashes between a subclass instance and a superclass instance are always resolved in favour of the subclass. Maps allow us to specify objects as environments; multi-layer regions allow us to specify the inheritance hierarchy directly by providing a facility to "shadow" names defined in different classes found along an inheritance chain.

Consider a simple example: `rectangle` is a subclass of `graphics-object`. `Graphics-object` defines two methods: `set-color` and `set-name`, and defines two instance variables: `color` and `name`. `Rectangle` is a class that defines four methods: `set-width`, `set-height`, `set-name` and `area`, and defines three instance variables: `width`, `height` and `name`.

```
rectangle :
  (map
     make-rectangle : (lambda ()
                          (layer (rectangle-instance-vars)
                                 rectangle-methods
                                 (graphics-object.make-graphics-object)))
     rectangle-instance-vars : (kappa ()
                                   width : <default-value>
                                   height : <default-value>
                                   name : <default-value>)
     rectangle-methods : (map
                              set-width : (lambda (rectangle-instance new-width)
                                              ... )
                              set-height : (lambda (rectangle-instance new-height)
                                              ... )
                                              ... )
                              set-name : (lambda (rectangle-instance new-height)
                                              ... )
                              area : (lambda (rectangle-instance) ... )))
  graphics-object :
    (map
      make-graphics-object : (lambda ()
                                (layer (graphics-object-instance-vars)
                                       graphics-methods))
      graphics-instance-vars : (kappa ()
                                  color : <default-value>
                                  name  : <default-value>)
      graphics-methods : (map
                             set-color : (lambda (graphics-object-instance new-color)
                                            ...)
                             set-name  : (lambda (graphics-object-instance new-name)
                                            ...)))
```

The object returned by **make-graphics-object** is a map:

```
(map
  color : <default-value>
  name  : <default-value>
  set-color : (lambda (...))
  set-name  : (lambda (...)))
```

The object returned by **make-rectangle** then becomes the map:

```
(map
  width  : <default-value>
  height : <default-value>
  name : <default-value for name in rectangle> |
         <default-value for name in graphics-object>
  set-width  : (lambda (...))
  set-height : (lambda (...))
  set-name : <method in the rectangle class> |
             <method in the graphics-object class>
  color : <default-value>
  set-color : (lambda (...)))
```

Note that because maps are always shared, never copied, the maps defining *rectangle's methods* and *graphic-object's methods* are references to, not copies of, the methods defined in their respective classes. The objects returned by the **make-rectangle** and **make-graphics-object** functions differ from a typical Smalltalk object in that they contain a direct reference to the methods defined by their defining class; like Smalltalk, however, Symmetric Lisp's map-sharing semantics dictates that any change to a method be seen by all extant instances of the class in which the method is found.

Message-passing in this model is simply expression evaluation within the object; it is semantically no different from a record selection operation. If `my-rectangle` is bound to the above map, then the expression

```
  (my-rectangle.< exp >)
```

evaluates *exp* using the instance variables and methods defined in `my-rectangle`; if the identifier `name` or `set-name` is referred to in *exp*, then the value of identifier will be its value as defined by `rectangle`.

For example, evaluating `my-rectangle.height` returns the value bound to the `height` variable in object `my-rectangle`; the expression

```
  (my-rectangle.set-color my-rectangle "blue")
```

sets `my-rectangle`'s color to be `"blue"`. The expression

```
  (my-rectangle.set-name my-rectangle "foo")
```

applies the `set-name` method defined in `rectangle` to arguments `my-rectangle` and `"foo"`. The `set-name` method defined by `graphics-objects` is superseded with the one defined by `rectangle`.

The Smalltalk-style expression

```
  (send object method args)
```

is, therefore, represented in Symmetric Lisp as

```
  (object.method args)
```

Note that this is a *parallel* object-based system: many objects may send (and receive) messages to (and from) one another concurrently.

Multiple-inheritance systems are an extension of simple inheritance hierarchies in which an object can belong to several incomparable superclasses; whereas the subclass relation in a simple inheritance scheme is constrained to form a tree, the subclass relation in a multiple-inheritance system can form a directed acylic graph. One can express a simple form of multiple-inheritance in this model by flattening out the dependency graph at the time of object creation. For example suppose that `polygon-object` is also a superclass of `rectangle`, but is not a superclass of `graphics-object`. In other words, the rectangle class is to inherit the instance variables and methods from both `graphics-object` and `polygon-object`. If the `make-rectangle` function were redefined to be:

```
make-rectangle : (lambda ()
                   (layer (rectangle-instance-vars)
                          rectangle-methods
                          (polygon-object.make-polygon-object)
                          (graphics-object.make-graphics-object)))
```

then all name clashes between `graphics-object` and `polygon-object` that occur in the evaluation of messages sent to a rectangle instance are resolved in favour of `polygon-object`. This approach to handling multiple-inheritance is similar to the solution adopted by CommonLoops[14].

Symmetric Lisp views the inheritance problem as essentially a namespace management problem; the inheritance hierarchy specifies a namespace that is composed from a collection of environments that may define different bindings for the same name. Any system that allows the user to build an evaluation environment in this way can also support an inheritance-based program methodology. For example, the class construct of Simula and Smalltalk or the defflavor structure found in Zetalisp specify an evaluation environment in much the same way that a Symmetric Lisp map does. One can explictly build an environment chain (or tree) in Smalltalk (or ZetaLisp) using the notion of a superclass (or flavor); this ability is captured in Symmetric Lisp directly through layered regions that represent namespace hierarchies. One can build an inheritance hierarchy in Amber by viewing records as local environments and using subtype relations to position different records within the hierarchy. Symmetric Lisp, like Smalltalk and unlike Amber, does not employ type inference to determine inheritance relations among different objects; superclass/subclass relations are explicitly notated in the object template.

4.2 Modularity

The previous section described how a number of diverse, well-known program structures can be understood in terms of map definition and evaluation. The motivation for studying the symmetric model stems not just from our interest in understanding how different program forms can be modeled, but in our intuition that these structures can be synthesized to produce new program constructs and associated new paradigms. This section discusses one such *modular* program structure and its implementation in Symmetric Lisp.

Modularity refers to the synthesis of big structures out of separate smaller structures. Concretely, the idea involves constraints on information flow or access between pieces, which are most often named groups of named procedure definitions or object declarations. A particularly interesting kind of modularity is *recursive modularity*, in which smaller components are structurally the same as the elements of which they are a part. The symmetric programming model supports recursive modularity: maps may contain sub-maps nested to any depth as well; the structure of any component map is identical to the structure of the whole. Modularity in a symmetric language also implies parallelism: whenever two program pieces are hooked together, they execute concurrently. Most importantly, the modular, parallel program structure is in fact not just a program structure, but a data object that can be manipulated according to the normal rules for object manipulation.

Consider the following problem: given a large and complicated mass of input data arriving on different data streams, one requires a system that performs two separate but related tasks: (1) It will act as a heuristic database, ready to accept user queries either about the status of a particular data-stream or the likelihood of some more complex phenomenon given the current state. (2) It will act as a monitor and alarm system, posting notices when significant state-changes occur. In determining the likelihood of more complex states, the system might use quantitative tests, heuristic decision procedures (as rule-based systems do, for example), or any mix of the two. Systems of this sort have wide applicability in the field of heuristic monitors and databases.

One useful organizational paradigm for systems of this sort is the "cooperating experts" or "blackboard" model first used in the implementation of the Hearsay-II knowledge-based system[29]. In this paradigm, the application domain is divided into a number of specialized sub-domains, each of which is under the supervision of some expert. Experts are free to transmit hypotheses and results to one another and information transmitted by one expert may be shared by several others. The model lends itself to a simple parallel formulation; a cooperating experts-based program in which individual experts are viewed as long-lived, *concurrently executing* processes is referred to as a *concurrent knowledge daemon.*

A *parallel process lattice* is a natural organization structure for a concurrent knowledge daemon. A parallel process lattice is a hierarchical network of nodes; each node represents a concurrently executing process, and communication between nodes follows the edges in the network. It is assumed that the bottom rung in the network is wired directly to an external sensor, and performs initial data processing and filtering, *e.g.*, it may be connected to a terminal or tty-line, a tactile sensor, or a blood-pressure monitor. Processes at higher levels in the lattice represent more complex "experts" that are responsible for monitoring and analyzing more complex states. In a program to monitor patients in a post-operative cardiac intensive-care unit (ICU), for example, low-level nodes would connect to blood-loss and blood-gain devices; they and other bottom-rung nodes converse with a higher-level node that calculates total fluid volume, which communicates with a higher-level node monitoring the likelihood of hypovolemia (a particular clinically-significant state) and so on.

A process-lattice is a bi-directional communication structure. Data filter up; queries filter down. An intermediate-level node reads data from the nodes below it; if its own state changes as a consequence, it communicates its new state to its own superior nodes. Nodes report their status on the display only when something "interesting" has happened or is likely to. The user may on the other hand enter a query at any time: a query amounts in effect to a question of the form "what is the current status of node n, and why?" Say the user of an ICU knowledge daemon suspects the presence of cardiac tamponade, but the system is reporting nothing on the topic: the user queries the status of the "cardiac tamponade" node, which may itself lack recent data because its inferior nodes haven't recently changed state; it propagates the query downwards, and eventually reports to the user. This sort of structure has been designed and tested in Linda[19] on a proto-type basis for a small automobile-traffic monitor and a monitor for patients in an ICU, but the formulation in Symmetric Lisp differs from the Linda implementation in some important ways as described below.

The process lattice is a modular, parallel program structure. To implement it well requires support for multi-streams, long-lived processes with externally-visible state, processes representing environments within which queries made by external processes can be evaluated, and the capability to generate processes dynamically. (Recall that multi-streams are distributed data structures to which many processes can append and read from simultaneously.) Long-lived processes are object-like processes whose internal state can be examined freely and from the outside. Symmetric Lisp is a good language to implement such a structure: because a Symmetric Lisp program is in fact a data object, one can structure the lattice as a map in which each map element corresponds to some node in the lattice. Each node in the lattice is itself a

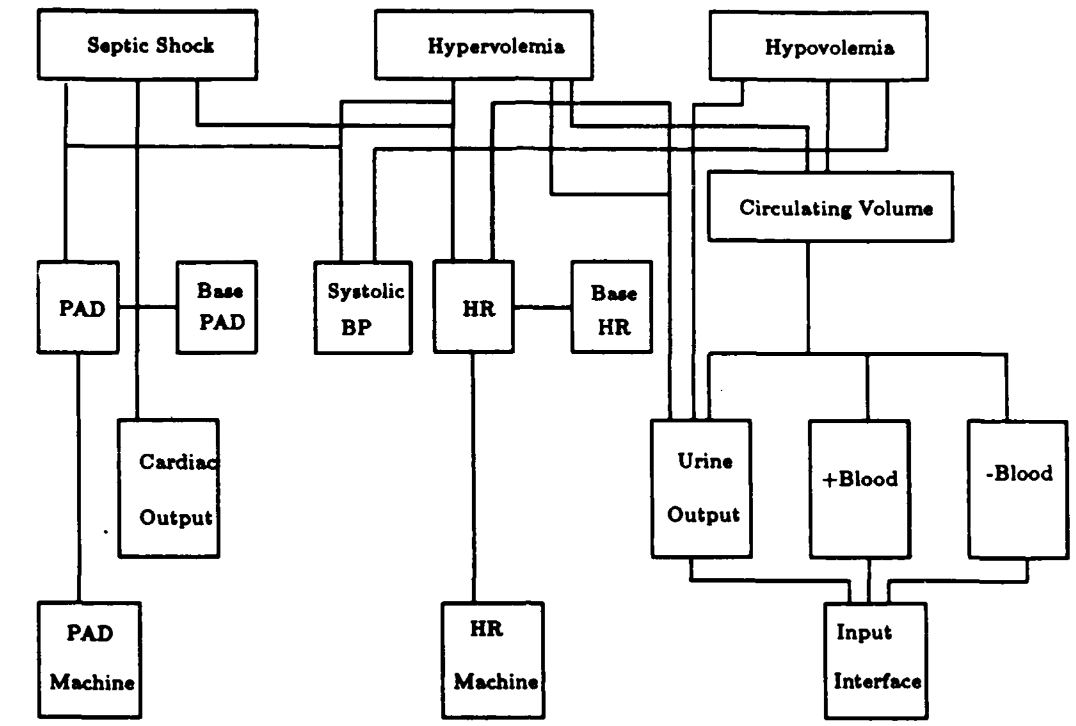

Figure 4 : A Subset of a Process Lattice for ICU Monitoring

map[9]; these nodes play a dual role: (1) they are perpetually active processes in the sense that they are constantly responding to queries and generating hypotheses based on current state information and (2) they are full-fledged data objects in the sense that their internal state may be examined by, and passed as arguments to, other processes.

4.2.1 The Structure of a Process Lattice

Consider the small subset of a process lattice for ICU monitoring shown in Figure 4.

The process lattice is a map:

```
proc-lattice : (map
                    systolic-bp : (map ...)
                    heart-rate  : (map ...)
                         ...
                    septic-shock : (map ...)
                    hypovolemia  : (map ...)
```

[9]Hence, the process lattice exhibits recursive modularity.

```
                    hypervolemia : (map ...))
```

What should be the structure of a lattice node? To answer this question, one must first determine how data and queries are communicated between nodes. Inter-node communication should be as flexible and transparent as possible; in a communication-bound system like the process lattice, rigid or cumbersome protocols that require nodes to format complex query and data messages would make for an especially inefficient and unappealing program structure. Efficient realization of the process lattice requires an inter-process communication model in which data and query transmission is highly uniform and transparent; ideally, communication should involve little, if any, synchronization between node processes. The open-map abstraction discussed in Section 2.2 defines a flexible communication structure since it can be treated as a multi-stream.

One useful way of structuring communication in a process lattice is to have superior nodes directly examine the state of an inferior upon notification that a state change has taken place. When a new data value is produced, a node attaches a signal to a local multi-stream indicating this fact; monitors found in nodes at higher levels of the lattice are then free to recompute a new state for their node based on the newly updated states of their inferiors. Each node map contains the following fields:

```
      ...
query-map : (open-map.create)
new-data?  : (open-map.create)
      ...
```

To notify a superior that a new state has been computed, an inferior executes

```
(attach new-data? t)
```

Because the contents of a map can be examined from the outside, superior nodes can directly examine the state of their inferiors without the inferior's knowledge. Since inferiors are unaware of who their superiors are, one can dynamically add new nodes on top of the lattice without altering the existing structure. Assuming that each node maintains a variable, **state**, a superior node can find the current state value of an inferior *i* by evaluating *i*.**state**.

To send a query down to its inferiors, a node would execute

```
(onto (lambda (inferior-node)
        (attach inferior-node.query-map t))
   inferiors)        ;;; inferiors is a map of the node's inferiors
```

A query is basically a signal indicating that a superior wishes the subordinate to compute a new state. The query-map is under constant surveillance; when a query is received, the inferior node propagates the query down to its node's inferiors. A node that has no inferiors simply recomputes a new-state and attaches a signal to its `new-data?` map. One can structure such a monitor as follows:

```
query-monitor : (lambda (query-map)
                  (seqmap
                     query : (mcar query-map)
                     (if (empty? inferiors)
                         (compute-new-state)
                         (onto (lambda (inferior-node)
                                     (attach inferior-node.query-map t))
                             inferiors))
                     (query-monitor (mcdr query-map))))
```

The `data-map` monitor evaluates a new state based on the updated information of its node's inferiors. The structure of a data monitor might look something like

```
data-monitor : (onto (lambda (inferior-node)
                         node-monitor : (lambda (new-data?)
                                            (seqmap
                                               new-node-state : (mcar new-data?)
                                               (compute-new-state new-node-state)
                                               (node-monitor (mcdr new-data?))))
                         (node-monitor inferior-node.new-data?))
                   inferiors)
```

The `compute-new-state` function computes a new-state and attaches a signal onto the local `new-data?` map.

The data monitor is actually a map of monitors with each sub-monitor dedicated to surveying state changes in one of the node's inferiors. A sub-monitor waits for its particular inferior to compute a new state value and then invokes the node's decision procedure, `compute-new-state`, passing the newly updated inferior node as the argument. Sub-monitors evaluate in parallel; monitors from different nodes can simultaneously examine the same inferior.

Programmers have great latitude in implementing a node's decision procedure. Upon receiving word that an inferior has changed state, a node may immediately compute a new state, query some (or all) of its other inferiors to recompute their states, or it may choose to ignore the new information entirely. Suppose that each node shares access to a global system clock maintained in a `time` map. Each node records the time of its last update in a `update-time` field. When a decision procedure is invoked, it may choose to query other nodes if they have not been active recently. For example, if the decision procedure for `hypovolemia` is invoked because of new data produced by the `systolic-bp` node, it may choose to query the `hr` node if no new data on heart rate has been recently received:

```
...
(if (> time.current (+ hr.update-time delta))
    (attach hr.query t))
...
```

`time.current` is the global time; `hr.update-time` is the time at which the heart rate was last noted and `delta` is a tolerance factor. The heart rate node will recompute its state upon receiving the query. The hypovolemia process monitoring heart-rate will subsequently invoke the hypovolemia decision procedure when a new heart-rate is computed.

The non-strict map evaluation rule makes it particularly convenient to monitor the evolution of objects of various kinds. For example, suppose that the user wishes to record how many times in total a node changes status; one could evaluate the following expression to perform this task:

```
state-monitor :
  (generate (i proc-lattice)
     (map
        state-counter : 0
        record-change : (lambda (new-data?)
                           current-state : i.state
                           (seqmap
                              new-state? : (mcar new-data?)
                              (if (<> current-state new-state?)
                                  (set state-counter (1+ state-counter)))
                              (record-change (mcdr new-data?))))
        (generate (j (iota 1 (msize proc-lattice)))
             (record-change proc-lattice[j].new-data?))))
```

Because the process lattice has spatial characteristics, one can iterate over its elements as one would an array or list. The `state-monitor` is a map containing a map for each node in the lattice. Each component map consists of two fields: a state counter and a daemon function that waits for a state change on the node being monitored; whenever a change occurs, the counter for that node is incremented. As was the case in the previous example, this monitor also runs indefinitely and can examine the behaviour of the process lattice and the evolution of lattice nodes without disturbing the core program. (Of course, monitoring the behaviour in *realtime* requires that the underlying scheduler guarantee that a monitor process will not miss a state change because of scheduling constraints. We don't address the process scheduling issues here.)

The dynamic nature of the process lattice is especially useful given that it is intended to operate over a long period and in an interactive setting. For example, suppose the user wishes to monitor the values of septic shock, hypovolemia and hypervolemia using some new higher-level decision procedure. He would evaluate the expression,

```
(f proc-lattice.hypovolemia
   proc-lattice.hypervolemia
   proc-lattice.septic-shock)
```

where *f* is the decision procedure. *f* may be a monitor that looks like

```
f : (kappa (hypovolemia hypervolemia septic-shock)
       ( loop forever monitoring for interesting results))
```

Once installed, **f** runs indefinitely, informing the user whenever some particular combination of these three manifestations occurs. Because *f* is represented as a kappa, and kappas expand upon application into maps, *f*'s application can be under evaluation even as other expressions are attached to the interpreter. The process lattice structure itself is unaware of the existence of *f* and need not be altered to accomodate it. The ICU process lattice *program* can be examined from the outside (by *f*) precisely because it has the semantics of a *data* object. Notice that Symmetric Lisp's support for recursive modularity makes it possible to incorporate the lattice program as an element of an ICU-interpreter environment; thus, we can potentially have multiple incarnations of different process lattices monitoring different patients:

```
ICU-interpreter : (map
                      proc-lattice-1 : (map
                                           systolic-bp : ...
                                           heart-rate : ...
                                              ...)
                      proc-lattice-2 : (map ... )
                            ...
                      proc-lattice-k : (map ... ))
```

The **ICU-interpreter** encapsulates *k* process lattices; each lattice would be responsible for monitoring different input streams.

How easily can the process lattice be implemented in other modular, parallel languages? Parallel languages such as Ada or CSP implement process communication using explicit rendevous points and do not support dynamic process instantiation. By requiring communicating processes to synchronize at a specified send/receive entry point, an Ada or CSP program cannot implement the asynchronous communication behaviour of the Symmetric Lisp version of the process lattice. Unlike a distributed data structure scheme that allows the programmer to decouple the task of process creation from the job of process communication, every Ada task (or CSP process) must know the identity of all other tasks (or processes) with whom it needs to communicate at the time that it is defined. The requirement that programs must exhibit a static process structure also prevents the lattice from being monitored by dynamically-created external processes.

The Symmetric Lisp implementation differs from the Linda one in one major respect. Because a Linda process is not a data structure, its internal state is not visible for inspection by other processes; the elements of a process state which are to be made visible to other processes must be dropped explicitly into tuple-space. Thus, unlike a Symmetric Lisp map, a Linda process whose state is to be visible to the outside must inject the necessary elements into tuple-space where they may be subsequently read; Linda does not support *transparent* access to a process state.

Non-strict parallel languages such as MultiLisp[40], Id[7] or SAL[4] can presumably also be used to implement the a process lattice. A process lattice node would be implemented in MultiLisp using futures, in Id using managers[5], and in SAL using Actors. Process communication in MultiLisp is achieved using futures and semaphores; Id expressions communicate results to other concurrently executing expressions via the internal state held by a manager; message passing is used in SAL to implement process communication. Because neither a MultiLisp future, an Id manager, or a SAL actor, however, is a data structure in the sense that a Symmetric Lisp map is, we would expect the formulation of the process lattice in these languages to be quite different from the one given here.

4.3 Maps as Parallel Program Objects

The evaluation model for maps we have considered in this thesis treats a map program a parallel program in which each region defines a separate parallel thread. Names declared in a map serve as synchronization points; an expression requiring the value of a name synchronizes with the expression producing the name's binding-value. This section examines the role of maps as parallel program structures; the discussion is primarily concerned with explicit concurrency of the kind found in parallel simulations or resource management applications.

4.3.1 Pipeline-Based Prime Number Generation

As a first example, consider a pipeline-based prime number generator. In such an algorithm, we represent the prime number filtering process as a pipeline in which each pipeline component represents a process dedicated to filtering out all multiples of a particular prime number; each successive component of the pipeline sees only those elements that are not multiples of primes seen earlier in the pipe. Whenever the current last component in the pipeline sees a number that is not a multiple of the prime it represents (*i.e.*, whenever it encounters a new prime), it extends the pipeline by adding a new component process responsible for filtering out multiples of this new prime.

This formulation is basically a systolic-style implementation of the standard sieve of Erathosthenes solution. In the abstract, we can consider a systolic program as program with a particular shape: a parallel algorithm is embodied in a graph that captures an efficient traffic pattern of data and results, and the program itself embodies the graph. In the prime number generator given below, *each element of the pipeline represents a systolic program module: data is transmitted along the pipeline in the same way that information is transmitted along the components of a systolic array.*

The code for the sieve is given below:

```
primes : (lambda (size n)
   prime-list : (generate (i (iota 1 n)) *)
   (fill prime-list[1] t)
   (fill prime-list[2] t)
   make-integer-stream : (lambda (m n)
     int-stream : (open-map.create)
     loop : (lambda (stream m)
              (if (> m n)
                  stream
                  (seqmap
                     (attach stream m)
                     (loop stream (1+ m) n))))
     (loop int-stream m))
   pipeline-element :
     (lambda (my-prime sieve)
        outbox : (open-map.create)
        end-of-pipe : t
```

```
          loop : (lambda (sieve)
                   next-candidate : (mcar sieve)
                   (if (multiple? next-candidate my-prime)
                       (seqmap
                         (fill prime-list[next-candidate] nil)
                         (loop (mcdr sieve)))
                       (if end-of-pipe?
                           (seqmap
                             (fill prime-list[next-candidate] t)
                             (pipeline-element next-candidate outbox)
                             (set end-of-pipe nil)
                             (loop (mcdr sieve)))
                           (seqmap
                             (attach outbox next-candidate)
                             (loop (mcdr sieve))))))
        (loop sieve))
      (pipeline-element 2 (make-integer-stream 3 size))
      prime-list)
```

The function **primes** takes as its input two integers, the first indicating the size of the integer sequence to be examined and the second indicating the number of primes desired in that sequence. (We assume that **size** > **n**.) It defines three bindings: the first is the **prime-list** whose i^{th} element will be t if i is prime and nil otherwise; the second binding defines an integer stream producer that, given integers m and n, returns an open-map containing elements m through n; the third binding defines the **pipeline-element** function. This function implements the sieve. It defines three bindings: an outbox which represents the elements *not* filtered by this process (*i.e.*, the elements that are not multiples of the prime this process represents), an **end-of-pipe** flag that is true if this process is the last element in the pipeline, and false otherwise and a loop function. The loop function scans down the input sieve; if the first number in the sieve is a multiple of its prime, it continues looping; if it is not, then there are two possibilities. If the loop is part of an element that happens to be the last component in the pipeline, then the number it has just scanned must be a prime; in this case, it records the fact in the **prime-list**, adds a new pipeline element whose filter agent is the prime number just scanned, sets its own **end-of-pipe** flag to nil and, finally, recurses (in that order). If the function is not part of the last pipeline element, it attaches the scanned number to its outbox and recurses; the loop function found in the pipeline element to the right of this component uses this outbox as its sieve.

We illustrate this process in Figure 5. The list of primes in the input set are those elements of **prime-list** with value t. Note that we haven't considered termination of the loop processes;

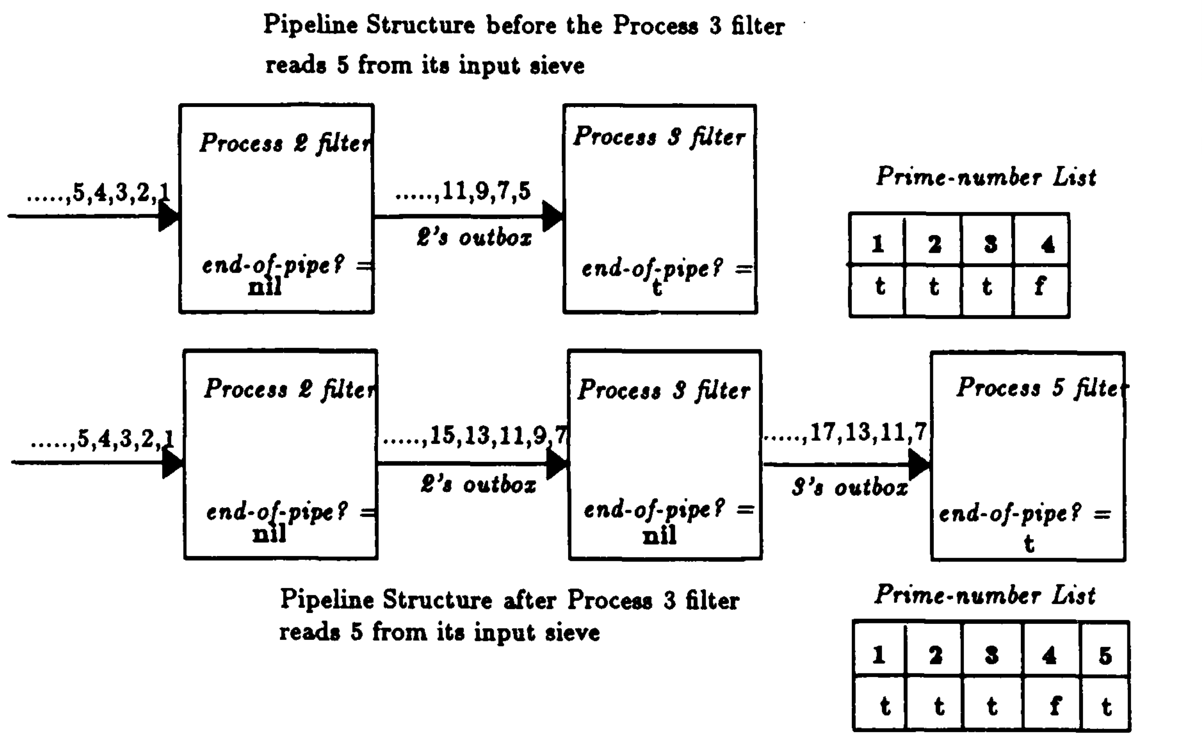

Figure 5 : Pipeline-Based Implementation of a Prime Number Generator after all elements in the input set have been processed, each loop process will eventually quiesce waiting for a new number to be attached to its input sieve. To terminate the loop processes, we would have to add a termination signal to each input sieve.

The function returns the `prime-list` as its result; the caller can examine whether a given number is prime or not by using the number to index into the map returned; the semantics of empty regions will cause the accessing expression to block until a value (either `t` or `nil`) is filled in by the appropriate process.

There are a number of simple variations on this program that are easily accomodated. We could have, for example, designed the program so that it generates *all* primes in the given list. To get this behaviour, we would have represented the `prime-list` as an open-map (rather than a flat map of empty-regions). Whenever a prime is encountered, it would be attached to this list. The basic loop structure of the `pipeline-element` function would remain unchanged.

4.3.2 Implementing a Process Network

We expand upon the systolic-array style paradigm introduced in the previous example by examining the implementation of more general process networks. Consider how one might express the Hamming numbers problem in Symmetric Lisp – the task is to print in ascending order all numbers of the form $2^a * 3^b * 5^c$, for $a, b, c \geq 0$. One nice formulation of this problem is in terms of a network of communicating processes. The network, shown in figure 6, consists of five nodes: two merge gates and three stream generators. The three generators simply multiply the elements on the output stream by a constant (either 2,3 or 5) and feed the result back to one of the merge gates. The merge boxes take two streams as arguments and produce an ordered stream as their result. It is not difficult to convince oneself that the elements on the output stream generated by the second merge gate (which is also the input stream for the three stream generators) is an ascending stream of Hamming numbers.

Can one implement a cyclic network of this kind in Symmetric Lisp? A simple and elegant solution to this problem can be expressed in functional languages supporting infinite lists; the solution to the Hamming problem in Miranda can be expressed succintly as follows:

```
hamming = 1 : merge (f 2) (merge (f 3) (f 5))
          where
               f a = [ n*a | n <- hamming ]
               merge (a:x) (b:y) = a : merge x (b:y), a < b
```

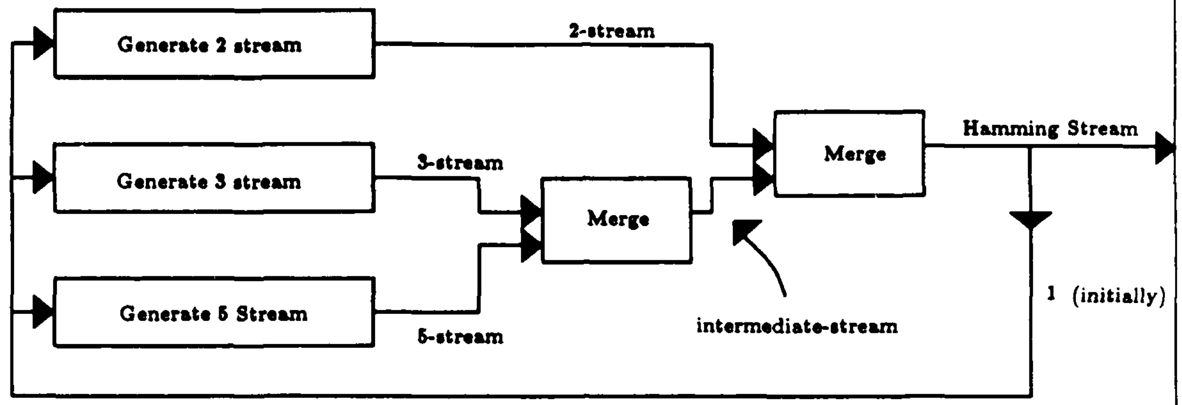

Figure 6 : A Network to Implement the Hamming Numbers Problem

```
                    = b : merge (a : x) y, a > b
                    = a : merge x y, a = b
```

It is possible to translate the Miranda-style solution into an equivalent (functional) Symmetric Lisp program either using the **generate** comprehension form or by building stream generators (in the style given in Section 4.1.5) for the various streams defined by the different applications of **f**. Another possibility, and the one we examine here, is to build the streams explicitly as we did in the prime number example.

As in the previous example, we can use open-maps to represent the communication channels; producers and consumers automatically synchronize because of the semantics of empty regions. Because the language contains facilities for explicitly synchronizing concurrently executing processes via data structure access, it is possible to implement a deadlock-free cyclic network of this kind.

The basic idea in the Symmetric Lisp implementation is to represent streams as open-maps that are extended by appropriate stream generators which monitor the presence of new elements on the output stream.

```
ham : (lambda ()
   2-stream : (open-map.create)
   3-stream : (open-map.create)
   5-stream : (open-map.create)
   hamming-stream : (open-map.create)
   intermediate-stream : (open-map.create)
   (attach hamming-stream 1)
multiply : (lambda (stream)
              (seqmap
                 (map
                   (attach 2-stream (* 2 (mcar stream)))
                   (attach 3-stream (* 3 (mcar stream)))
                   (attach 5-stream (* 5 (mcar stream))))
                (multiply (mcdr stream))))
(multiply hamming-stream)
merge : (lambda (stream-1 stream-2 output-stream)
           hd-1 : (mcar stream-1)
           hd-2 : (mcar stream-2)
           (cond ((= hd-1 hd-2)
                  (seqmap
                     (attach output-stream hd-1)
                     (merge (mcdr stream-1) (mcdr stream-2) output-stream)))
                 ((> hd-1 hd-2)
                  (seqmap
                     (attach output-stream hd-2)
                     (merge stream-1 (mcdr stream-2) output-stream)))
                 ((< hd-1 hd-2)
                  (seqmap
                     (attach output-stream hd-1)
                     (merge (mcdr stream-1) stream-2) output-stream))))
 (merge 3-stream 5-stream intermediate-stream)
 (merge 2-stream intermediate-stream hamming-stream)
 hamming-stream)
```

The `multiply` function acts as a stream multipler that multiplies all elements of the hamming stream by 2, 3 and 5, attaching the results to the appropriate output stream (either `2-stream`, `3-stream` or `5-stream`). The value of the hamming stream is determined by the merge of `2-stream` with the merge of `3-stream` and `5-stream`. The `merge` function does a sort on the elements of its two input streams attaching the elements of the sort to its third argument. By enclosing the `attach` operation and the recursive call inside a `seqmap` when performing a merge, we guarantee that the ordering of the output stream is consistent with the relative order of the input streams.

Process synchronization takes place whenever a stream multiplier (in the `multiply` function) or a merge operator executes an `mcar` operation. If the element referenced is an empty region, the operation suspends until a new element is attached. In the case of a multiplier, it is only when a new element is added to the hamming stream that the multiplication proceeds and a new element is attached to the specific multiplier stream; in the case of the `merge` function, it is only when a new element is added to the appropriate multiplier stream that the hamming stream is extended.

4.3.3 A Dataflow Simulation

We now consider a further generalization of the two previous examples. We would like to consider an emulation device for static dataflow graphs[10]. The Symmetric Lisp emulator is to be faithful to the semantics of the dataflow actors found in the static dataflow graph with respect to the synchronization and firing rules they obey.

There are two reasons why Symmetric Lisp's synchronization and process semantics make it a good tool for modeling concurrent systems. First, the non-strict semantics of maps means that lightweight, long-lived daemon processes are natural organization structures. Secondly, the fact that processes can read and write common data streams using empty regions as the basic sychronization mechanism means that one can directly model general producer/consumer relationships; in particular, one can directly model the dataflow dependencies that exist between actors in a dataflow graph.

Nodes in the abstract dataflow graph are represented as perpetually running process monitors and edges are implemented as open-maps. Each process watches the open-maps corresponding to the input edges for the node it represents and computes a result based on input values found in these map streams. These results are then written to the open-map corresponding to the node's output edge. Monitors execute asynchronously (in the same way that actors in a real dataflow system do). In the particular translation given here, acknowledgement arcs between nodes are not used: data written onto open-maps are queued; the translation guarantees that the order in which output values are emitted by a node is preserved when writing onto the appropriate edge by explicitly serializing the writing of an output value with the reading of new inputs[11].

Consider the following program fragment written in Val[2] to compute the factorial of a number:

```
Function Factorial (n : integer returns integer)
    for i : integer := 0;
        p : integer := 1;
    do if i = n then p
         else
             iter i := i + 1;
```

[10]For our purposes, a static data flow language is one in which the structure of the base language graphs is fixed at compile time; there are no function application operators that can instantiate new copies of function graphs at runtime. Iteration is supported by allowing cyclic graphs to be constructed. Readers unfamilar with the dataflow model of computation should consult [6] which gives a comprehensive introduction to the subject.

[11]In other words, this simulation assumes unlimited queuing on edges.

```
                $p := p * i$
            **enditer**
        **endif**
    **endfor**
**endfun**
```

The `iter` construct creates a new "local" environment for `i` and `p` and evaluates the expressions to which they are bound in the context of this new environment.

A possible translation of this function into an abstract dataflow graph representation is shown in Figure 7. Edges entering into the sides of actors are signals – they generate boolean tokens. TRUE and FALSE gates pass their input only if the current value on their signal line is either **true** or **false**, resp.; they consume their input otherwise.

The corresponding representation in Symmetric Lisp is given below:

```
factorial :
  (lambda (n)
     i : (open-map.create) (attach i 0)         ; variable i.
     p : (open-map.create) (attach p 1)         ; variable p.
     answer : *
     edge1 : (open-map.create) ; link between = and true and false gates.
     edge2 : (open-map.create) ; link between first false gate and +.
     edge3 : (open-map.create) ; link between second false gate and *.

     actors : (map
               =-actor       : ((lambda (i p)
                                   (seqmap
                                      (if (= (mcar i) (mcar p))
                                          (attach edge1 "true")
                                          (attach edge1 "false"))
                                      (=-actor (mcdr i) (mcdr p))))
                                 i p)
               true-gate     : ((lambda (p edge1)
                                   (seqmap
                                      (if (= (mcar edge1) "true")
                                          (fill answer (mcar p)))
                                      (true-gate (mcdr p) (mcdr edge1))))
                                 p edge1)
               false-gate-1 : ((lambda (i edge1)
                                   (seqmap
                                      (if (= (mcar edge1) "false")
                                          (attach edge2 (mcar i)))
                                      (false-gate-1 (mcdr i) (mcdr edge1))))
                                 i edge1)
               false-gate-2 : ((lambda (p edge1)
                                   (seqmap
```

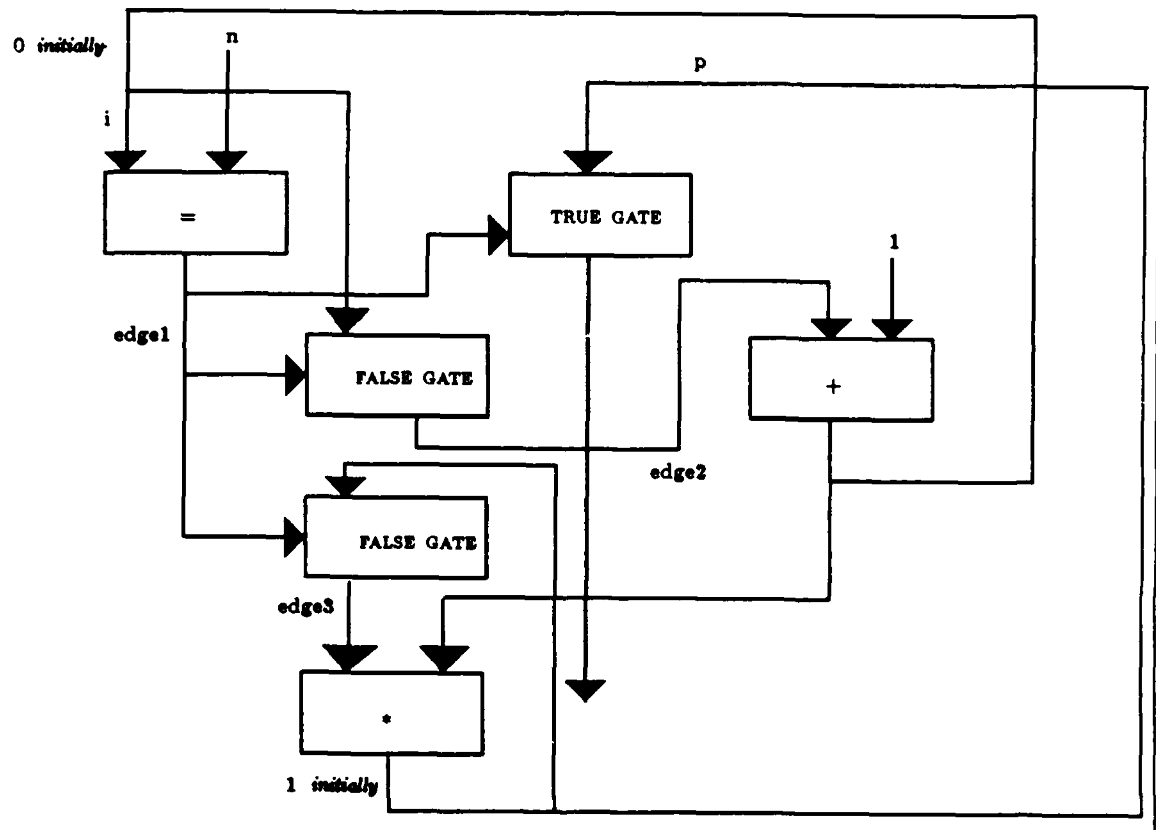

Figure 7 : A Static Dataflow Graph for Factorial

```
                                       (if (= (mcar edge1) "false")
                                          (attach edge3 (mcar p)))
                                       (false-gate-2 (mcdr p) (mcdr edge2))))
                                 p edge1)
                   +-actor     : ((lambda (edge2)
                                     (seqmap
                                        current-i : (mcar edge3)
                                        (attach i (1+ current-i))
                                        (+-actor (mcdr edge2))))
                                 edge3)
                   *-actor     : ((lambda (edge3 i)
                                     (seqmap
                                        (attach p (* (mcar edge3) (mcar i)))
                                        (*-actor (mcdr edge3) (mcdr i))))
                                 edge4 i))
        answer)
```

The translation of `factorial` into a Symmetric Lisp-style dataflow graph representation is naive and could be optimized considerably. We could have, for example, abstracted the code for the `false` gates into a lambda expression:

```
   false-gate : (lambda (input signal result)
                   (seqmap
                      (if (= (mcar signal) "false")
                         (attach result (mcar input)))
                      (false-gate (mcdr input) (mcdr signal))))
```

`false-gate-1` could now be bound to the application:

```
   (false-gate i edge1 edge2)
```

`false-gate-2` could be rewritten similarly. In effect, one could have written a library of actor templates that could be simply invoked with the appropriate arguments as defined by the structure of the graph.

Every edge in the abstract graph translates into an open-map; every actor translates into a daemon process. Each daemon process executes the same process repeatedly: (1) wait for new input, (2) attach result to the output stream and (3) recurse waiting for the new input tokens. For example, the `+-actor` waits for a value (a new value for `i`) to be attached to `edge2`, its input arc, by `false-gate-1`. Once a value is written, it increments it (this corresponds to the statement $i := i + 1$ in the Val program) and attaches the result to the output stream corresponding to `i` and waits again for new input. Despite the fact that all actors and gates can run asynchronously, the serialization introduced by the `seqmap` form guarantees that an output will not be produced until the corresponding inputs are received. Moreover, because every edge has only one producer, it is easy to see that merging of output values from different nodes cannot occur. The final result is given by the true gate which evaluates the `fill` expression that stores the result into `answer`. The true gate is activated only when all iterations have completed, *i.e.*, only when `i` = `p`.

How expensive is it to create an open-map for each edge? An open-map is a three-element structure; operations on open-maps are shared among all its instantiations. Viewed in this light, we conjecture that open-maps are not an expensive representation for edges in the dataflow graph. Of course, this conjecture can be validated only by experimentation which we leave as a topic for future investigation.

Note that the names introduced in the Symmetric Lisp version of the Val program are not strictly necessary; a mechanical translator could just as well have substituted index references for the name references introduced above. For example, all references to the open-map, `edge1`, could be replaced with the expression `(apply-env)[3]`. (Recall that `apply-env` returns the map address in which it is evaluated; in this example, this would be the map generated by the application of the factorial function. The third element in the application map is the open-map bound to the name `edge1`.) This same transformation would be applicable to each named actor as well. So far as the translator is concerned, the Symmetric Lisp map is simply a data structure containing open-maps, functions and evaluating expressions, all of whom can be addressed by their map index. The translation methodology sketched here is applicable to not just dataflow programs: any concurrent system (*e.g.*, synchronous systolic array programs, message-passing systems, RPC-based models, etc.) could be modeled in the same way so long as the necessary synchronization constraints and evaluation rules of the source language expressions are well-understood.

4.3.4 Resource Management

It has long been known that arbitrary side-effecting computation in a parallel programming language can lead to uncontrolled non-determinism. While unrestricted use of non-determinism makes it impossible to effectively reason about the behaviour of programs, constrained use of non-determinism can lead to novel and interesting applications. Several of the examples presented in this chapter, most notably the process lattice, depend crucially on the ability to write non-determinate code. Non-determinism is also especially useful in expressing resource manangement problems. A resource manager controls access to one or more resources shared by concurrently executing processes. Database transaction systems or operating systems are two domains where resource managers are typically required.

The basic approach to writing a resource manager in Symmetric Lisp is straightforward: define a number of daemon processes that constantly monitor a multi-stream to which users of the resource attach their requests; all daemon processes have access to the same data. The order in which requests are attached to the request multi-stream is non-determinate since it depends on the relative execution speeds of the active user processes. Updating of shared data takes place by first locking the appropriate structure before actually performing the update – this guarantees that no other daemon process can access the data during the period the update is being performed.

To illustrate, consider the following simple resource management problem[5]. A printer device is to be shared among many users. Jobs sent to the printer are either **slow** or **fast**. Fast jobs (jobs of less than 10 pages) have higher priority than slow jobs (jobs of greater than or equal to 10 pages): a slow job is printed only when there are no pending fast jobs to be printed. A job will be a two-element map:

```
(map
   ack : ⋆
   print-job : < text to be printed>)
```

When a job is printed, a notification is written onto the empty-region denoted by **ack**.

There are three multi-streams used to hold jobs:

```
input-stream :   (open-map.create)
slow-job-queue : (open-map.create)
fast-job-queue : (open-map.create)
```

one variable to act as a signal indicating when the **fast-job-count** goes to zero:

```
zero-fast-job : t ;  initially, there are no fast jobs
```

and two condition variables for slow jobs:

```
wait-for-zero-fast-job : nil ;  initially there are no fast jobs to be finished
zero-fast-job-signal : (open-map.create)
```

input-stream contains the job requests made by users; a user executes:

```
(attach input-stream < my-job>)
```

whenever he wishes to print a new job. To receive an acknowledgement that the job has been printed, the user tries to read *my-job*.ack.

The slow-job-queue and fast-job-queue contain the slow jobs and fast jobs (resp.) still waiting to be printed.

The number of fast jobs pending is held in a counter, fast-job-count, that is initially set to 0. In addition, the print manager also defines two locks, a print-lock which controls access to the printer and a fast-job-lock that controls access to the counter.

The split daemon partitions jobs found on the input stream by appending incoming jobs into either the slow job or fast job queue:

```
split : (lambda (input-stream)
          (seqmap
             next-job : (mcar input-stream)
             (seqmap
               (if (fast-job? next-job)
                   (add-fast-job next-job)
                   (add-slow-job next-job))
               (split (mcdr input-stream)))))
(split input-stream)
```

The daemon runs forever, waiting for new jobs to be added to the input stream. Whenever a new job is attached to the input stream by a user, it is either added to the slow job queue or fast job queue depending on its size. Note that both the recursive call and the call to the routine to add the new job to the appropriate queue can take place in parallel because they are enclosed within the same map. Thus, while old jobs are being processed, the split function is ready to accept and initiate processing of new ones. Note that this solution assumes a fair scheduler; in the absence of one, it is possible that a job may never be added to any queue, always overtaken by jobs introduced subsequently. The outer seqmap is necessary to throttle the unfolding of split activations; if it were not present, an unbounded number of activations of split would be immediately spawned when the function is initially applied (recall that the body of a function activation evaluates in parallel with the evaluation of the activation's arguments).

The add-fast-job and add-slow-job functions maintain the fast-job-queue and slow-job-queue resp.

```
add-fast-job : (lambda (job)
                  (seqmap
                     (holding fast-job-lock
                             (seqmap
                                (set fast-job-count (1+ fast-job-count))
                                (set zero-fast-job nil)))
                          (attach fast-job-queue job)))
   add-slow-job : (lambda (job)
                       (attach slow-job-queue job))
```

The add-fast-job function is responsible for incrementing the fast-job-count variable. It does this by first seizing the fast-job-lock, ensuring that only one extant activation can increment the counter at a time, and then attaches the job to the queue.

There are two daemons responsible for monitoring jobs added to the two queues:

```
   print-fast-job : (lambda (fast-job-queue)
                         (seqmap
                            job : (mcar fast-job-queue)
                            (holding print-lock (print job))
                            (holding fast-job-lock
                               (seqmap
                                  (set fast-job-count (1- fast-job-count))
                                  (if (= fast-job-count 0)
                                      (seqmap
                                        (set zero-fast-job t)
                                        (if wait-for-zero-fast-job
                                            (attach zero-fast-job-signal t))))))
                            (fill job.id "Job Printed")
                            (print-fast-job (mcdr fast-job-queue)))
   (print-fast-job fast-job-queue)
   print-slow-job : (lambda (slow-job-queue zero-fast-job-signal)
                         job : (mcar slow-job-queue)
                         (holding fast-job-lock
                             (if (not zero-fast-job)
                                 (set wait-for-zero-fast-job t)))
                         (if wait-for-zero-fast-job
                            (seqmap
                               (mcar zero-fast-job-signal)
                               (set wait-for-zero-fast-job nil)
                               (print-slow-job slow-job-queue
                                               (mcdr zero-fast-job-signal)))
                             (if zero-fast-job
                               (seqmap
                                  (holding print-lock (print job))
                                  (attach job.id "Job-printed")
                                  (print-slow-job (mcdr slow-job-queue)
                                                  zero-fast-job-signal))
                              (print-slow-job slow-job-queue zero-fast-job-signal))))
   (print-slow-job slow-job-queue zero-fast-job-signal)
```

The `print-fast-job` monitors the arrival of new fast jobs. When a fast job is ready to be printed, the daemon first seizes the printer (using the `print-lock` lock), decrements the `fast-job-count`, returns an acknowledgment message and recurses, waiting for the next job. If the number of fast jobs becomes zero, a signal is attached to the `zero-fast-job-queue`. If there are slow jobs waiting to be printed and their are no more fast jobs, the daemon attaches a signal to the `zero-fast-job-signal` queue.

The `print-slow-job` daemon is defined similarly except that it only attempts to print a job if there are no fast jobs waiting to be printed. If there are fast jobs still awaiting to be printed, it sets the `wait-for-zero-fast-job` flag indicating that a slow job is in the queue and waits for a signal to be attached by the `print-fast-job` daemon when no more fast jobs remain. When such a signal arrives, it resets the flag and recurses appropriately.

Note that this solution does *not* prevent starvation of slow jobs; a slow job is only printed after there are no more fast jobs remaining.

Although the example chosen is very simple, it nonetheless highlights some important issues in implementing resource managers. In particular, it illustrates the use of locks to guarantee mutual exclusion of shared data, the role of multi-streams as inter-process communication devices, and the structure of long-lived daemon processes. The solution is superior to a monitor-based one insofar as it allows daemon processes and multiple incarnations of the `add-fast-job` and `add-slow-job` to be active simultaneously within the same manager. The `print-slow-job` routine would be implemented in a monitor-based language using a condition variable that becomes true when there are no fast jobs to be printed; this functionality is subsumed by the `zero-fast-job` stream. In a manager[5] based solution, each multi-stream would have to be implemented as separate manager since non-determinate streams are not a user-visible datatype in a stream-based functional language.

4.4 Summary

The symmetric programming model was developed based on the hypothesis that superficially-diverse program and data structure forms found in conventional languages need not and should not be viewed as being distinct. The symmetric programming model is not an attempt to define an all-inclusive model for every programming language. Every real language constitutes at a minimum a series of refinements, embeddings and re-interpretations of the symmetric model; some languages, for example, Prolog[68] or Smalltalk[38], go significantly beyond or in different directions from it.

The symmetric model, does however, address some of the fundamental techniques in the management of program complexity. It does so by defining the organization of programs and data structures within a unified, *recursive* framework. The idea of recursive modularity - allowing sub-programs and data objects to have the same structural description as the programs of which they are a part – is central to our view of what symmetric programs represent and is a direct consequence of this unified treatment of programs and data. Recursive modularity is a specific example of the general symmetric metaphor: we now picture programs as cells, generally arranged linearly but occasionally in some other pattern. Each cell is structurally identical to every other cell, but certain sub-collections are specialized to particular duties – storing heterogenous, but related, values, holding the actual and formal parameters of a procedure activation, or holding entire programs. Even though the contents of these cells may differ, each cell relates uniformly to the rest: the same rules governing the meaning of names and position hold and the same evaluation rule applies across all cells. More importantly, each cell's structure is recursive: an arbitrarily-complex world can be installed inside any given cell.

In the introduction, we had argued that the fundamental characteristic of the symmetric programming model is its support for *uniformity* of various kinds. In this chapter, we examined solutions to problems that "asymmetric", "non-uniform" languages do not handle well. In the problems we examined, the conventional separation between program and data structure is not logically necessary, not only because it obscures basic similarities between these structures, but also because it forms a practical obstacle to realizing an efficient solution.

Our discussion fell into three categories: (1) describing continuity of program and data structure over module elaboration, (2) describing continuity of structures under change of viewpoint, and (3) describing continuity of structure in the presence of parallelism.

In exploring the first category, we observed that common program structures such as begin-end blocks relate very naturally to common data structures such as records or arrays and, furthermore, that both of these structures, in turn, are simple components of more complex objects such as libraries, packages or inheritance-based systems. Given the ability to select, abstract, and apply environments, we were able to formulate a simple, unified description of these superfically-diverse structures.

Our investigation of the second category involved a more complicated example that exploited the concept of recursive modularity; we described a process lattice *object* in terms of a *parallel program* consisting of sub-processes and sub-structures. Each of these sub-processes were themselves objects that could be examined via the usual operations available on data structures; each of these sub-structures were themselves programs that could be evaluated via the usual evaluation rule for program structures.

Finally, we examined the interaction between explicit parallelism and program/data uniformity. Our approach was to treat map structures as objects; in the symmetric model, whenever two map objects are hooked together, they form a parallel program. By hooking map structures together in various ways, we were able to derive a number of diverse parallel process structures ranging from systolic-based filters to dataflow subsystems. Here again, each process retained the semantics of a data structure and was manipulated as such.

In the next chapter, a more extensive example is developed. This example concerns the design of the Symmetric Lisp programming environment: how easy is it to implement a programming environment for Symmetric Lisp in Symmetric Lisp? The uniformity of program and data and support for first-class namespaces makes this a particularly interesting question.

Chapter 5

The Programming Environment

Symmetric Lisp is intended to serve as an interface to a highly parallel, interactive "meta-clean" computer system – a system in which virtually the entire state of the system is visible to and manipulable by the user. Symmetric Lisp also is intended to serve as an interface to a persistent computer – a system in which data objects can have lifetimes greater than the lifetime of the computations which created them.

How should interactive parallelism be supported in a monolingual programming environment? Contemporary research efforts in parallel programming and programming environments have largely ignored this question. This chapter argues that a uniform representation of programs and data in terms of first-class parallel environments provide the necessary ingredients to support a highly parallel, persistent[1] monolingual programming environment. The user of the Symmetric Lisp machine has direct control over all data structures that define his computing environment including the state of the file system. More importantly, this flexibility is achieved without introducing a separate system-interface command language or burdening the language with extra primitives to manipulate a language-external environment.

5.1 The Read-Layer Loop

The front-end (FE) of most interpreted languages is implemented by a `read-eval-print` loop – the FE acts as a virtual machine that repeatedly reads a new input expression, evaluates it on the basis of the internal environment structure maintained by the `eval` procedure, and prints

[1]It provides the necessary linguistic ingredients. The question of *how* to implement persistence is not addressed in this chapter.

the result. Users usually do not have access to the internal state of **eval** – programs to access and manipulate the environment image of an interpreter session must usually be provided as part of the evaluator package.

The role of the Symmetric Lisp front-end, on the other hand, is to implement a **read-layer** loop – expressions input by the user are added as a new element on top of the current environment; old bindings are superseded by new ones by layering the new binding expression on top of the old one; a map defines the environment image of the interpreter session. The map built by the front-end can be examined and manipulated as is. Because environments in Symmetric Lisp are governed by a parallel evaluation rule, the language supports interactive parallelism: input expressions in different layers of the growing map under construction by the Symmetric Lisp front-end are evaluated in parallel, up to the ordinary serialization rules imposed by the name evaluation rule.

The outline FE (ignoring issues of printing and formatting) can be written as follows:

```
FE : (lambda (user-env io-stream)
        (seqmap
           new-element : user-env.(read (mcar io-stream))
           (if (string= new-element[1] "end")
               user-env
               (FE (layer new-element user-env) (mcdr io-stream)))))
```

Expressions input by the user are represented as strings in an open-map called **io-stream**. The **read** operator reads the first element from **io-stream**, and coerces it into a map object whose evaluation environment is **read**'s dynamic environment[2]. **Read** is not a function that can be built out of Symmetric Lisp expressions described thus far; it is a special form that builds map objects from character strings.

Thus, in the above example, if **(mcar io-stream)** returned the string "**x : y**", the object returned by **read** would be:

```
(map x : y)
```

The evaluation environment of this map is the environment specified by **user-env**.

[2]**Read** plays the role of the **eval** operator found in most Lisps, and strings play the role of quoted objects. The effect of applying **eval** causes two levels of evaluation to occur on the argument form. First, the argument is evaluated as per the normal argument evaluation mechanism (which itself involves an implicit call to **eval**). The argument is then passed to the **eval** function where another evaluation occurs. In a similar sense, **read** takes a string and coerces it to produce an object that is then evaluated in the context of the **read**'s dynamic environment.

The free occurrence of `y` in this map is resolved by first searching for a binding-value in `user-env` and then in the lexical environment of `FE`. Note that the expression returned by `read` evaluates in whatever environment the `read` expression is evaluated.

If the object read is a termination string, the `FE` returns the environment image; otherwise, it recurses with a new environment that is built by layering the region contained in `new-element` onto `user-env`.

Because of the semantics of scope-expressions, no user-defined lookup or insert operations need to be performed on an interpreter-maintained environment structure.

A user creates a new interpreter session by creating a new map and invoking the front-end[3]:

```
new-env : (FE (map) io-stream)
```

The lexical-scoping rules of maps means that a function definition input to `user-env` only references the binding values of elements preceeding it in the map; elements input subsequently are not visible to the function (as per the normal scoping rules for maps). Suppose that `io-stream.stream` is structured as follows:

```
(map "y : 3"
     (map "f : (lambda (x) (+ x y))"
          (map "(f 2)"
               (map "y : 4"
                    (map "(f 2)" ★)))))
```

The corresponding structure of `new-env` after these expressions have been read (and evaluated) would be

```
(map   5                         ; result of second application
     y : 4                       ; the binding of y to 4 supersedes the old binding
       5                         ; result of first application
     f : (lambda (x) (+ x y)))   ; lambda definition
```

Because `lambdas` are lexically-scoped functions, rebinding `y` to 4 does not change the apply-time behaviour of `f`; changing the behaviour of `f` requires side-effecting `y`. If, instead of the binding declaration `y : 4`, the user input, `(set y 4)`, the second application of `f` would have used the new value of `y`. In other words, rebinding of definitions in an interpreter session does not change the behaviour of previously evaluated definitions; the only way to alter their behaviour

[3]Of course, there is usual bootstrap problem that has to be answered: who accepts these inputs in the first place, *i.e.*, under what interpreter-session does the front-end itself evaluate? We assume that there is a system-provided FE that is incarnated at the time the system is initialized.

is to side-effect the definitions that they do use. This property of the Symmetric Lisp top-level front-end is in contrast to the behaviour of most Lisps.

Because the front-end is an ordinary Symmetric Lisp function defined over an ordinary Symmetric Lisp environment, one can create multiple copies of it by invoking the function on different environment and input streams. Users can write routines that manipulate the environment image without having to alter the front-end code. Most language systems don't allow multiple incarnations of the front-end to be simultaneously active[4]. In systems which do (*e.g.*, Lisp Machines[39]), different interpreter sessions are constrained to execute within the same environment – it is not possible to instantiate different sessions (with overlapping lifetimes) that operate on different environment images.

MIT Scheme, by allowing first-class environments, makes it possible to create multiple versions of a meta-circular evaluator, but these evaluators must explicitly maintain and update the environment image – users in the middle of an interpreter session don't have access to the environment image maintained by the evaluator. (Users can examine the environment by coercing it into an alist object, but this is not the same as being able to examine and manipulate the environment image directly.) The evaluator image is buried within the meta-circular hierarchy: once the evaluator is invoked with an initial environment, all expressions input by the user are resolved only within the context of this environment. The structure of the evaluator prevents the user from directly accessing any of its internal structures.

5.1.1 User-Generated Namespaces

The Symmetric Lisp user/interpreter interface gives the user great latitude to build, share and manipulate naming environments interactively.

Consider the following example: suppose that a user while interacting with the system using E as the interpreter-session environment, wishes to redefine some definitions for test purposes. Ideally, he should be able to test these new definitions without having to touch the old versions. Because these new definitions may refer to definitions currently installed in E, they should be free to access E's elements without the user having to either change the front-end or the environment structure.

[4]While not a language-based system *per se*, Unix does allow multiple incarnations of the top-level shell to be simultaneously active; we discuss the relation between the Symmetric Lisp programming environment and a Unix-based one in Section 5.2.1.

Let the definitions to be tested be: D_1, D_2, ..., D_n. One first creates a new test environment, E_{test}, within E containing the new definitions:

```
E_test : (map D1 D2 ... Dn)
```

One can now proceed to test each of the D_i by typing:

```
(E_test.Di < args >)
```

to the front-end. This expression retrieves the test definition from the test environment; any free names encountered within D_i are resolved by searching first in E_{test} then in E. Thus, in testing D_i, any free references to D_j automatically resolve to D_j's test definition. One could evaluate a program p in the context of the environment defined by E_{test} by evaluating

$E_{test}.p$

All free references to any of the D_i found in p would resolve to their meaning in E_{test}.

By placing the new definitions inside a map, one effectively hides them from expressions subsequently input to E. If the i^{th} test definition is named D_i, the user can install it into the interpreter-session environment by writing

```
Di : E_test.Di
```

This binding is effectively installed as part of E.

Now, consider a generalization of the above example in which different interpreter-sessions need to (selectively) share information with one another. Suppose that E_1 and E_2 are the environment images for two incarnations of the Symmetric Lisp front-end:

```
E1 : (FE (map) io-stream1)
E2 : (FE (map) io-stream2)
```

Suppose that a user now wishes to instantiate a new interpreter session (call it E_3) in which free references to certain names are to be resolved using their values in either E_1 or E_2. For example, suppose that all free references to the name a occuring in E_3 need to be resolved using a's value in E_1 and all free references to name b need to be resolved using its value in E_2 (it is assumed that a and b are not redefined in E_3).

One might, of course, simply choose to copy the value of the relevant bindings into the new interpreter-session environment, but such a solution would not be appropriate if other active sessions also share (and update) E_1 and E_2. For example, if there is slightly modified incarnation of the FE that uses the environment defined by E_1 in evaluating its input expressions, changes made to E_1's bindings by E_3 would not be visible to expressions in this interpreter. Because of the possibility of sharing, then, a solution to this problem should not involve alterations to any of the existing interpreter environments. Shared namespace management of this sort is an important, but to-date largely ignored, issue in the design of multi-user language-based systems.

One can devise a solution to this problem in Symmetric Lisp by projecting the binding-values of a and b onto a new map that is then used as the interpreter-environment:

```
E3: (layer (select E1 a) (select E2 b))
```

E3 looks like:

```
(map a : < a's value in E1>
     b : < b's value in E2>)
```

One can now instantiate a new interpreter session thus:

```
(FE E3 io-stream)
```

where **io-stream** is an input stream as described earlier. Note that this solution also involves copying, but the semantics of map selection involves copying only addresses not structure values.

E_3 is a map that initially contains two regions: the first region contains a binding for a projected from E_1; the second regions contains a binding for b projected from E_2. Because the elements projected by the **select** operator are shared with the source maps, any changes made to a or b in the orignal maps will be visible to users of E_3. (It should be noted that this solution presumes that a and b are map structures; scalar objects are not shared between a map and its projections.) Assuming the definition of **FE** given earlier, the expressions found in the regions subsequently attached to this map, therefore, automatically evaluate in an environment that contains bindings for a and b. Moreover, any changes made to these names will become visible to other users of E_1 and E_2.

This solution required no alterations to either E_1 or E_2, nor did it involve annotating free occurrences of a or b in the input stream with particular environment prefixes. The ability to extend environments via the layer operation coupled with the map-sharing semantics of the name-projection operator are the novel properties of the language that allow such a formulation.

5.2 The Programming Environment

Most users of a computer system have very little control over the programming environment with which they interact. To manipulate their environment, they must either use the built-in utilities provided by the system implementors or write their own command language operations. While the command language may enable them to control their environment to a degree, it is usually not expressive enough for programming more general applications. Application languages, on the other hand, have historically been designed with little concern for the programming environment within which they are to execute; outside of providing some operations to read and write files, most languages isolate the user from gaining access to any other part of the command-level[5]. Consequently, users of a typical computer system must have fluency in at least two languages: they must be fluent in the command language to be able to run, modify and examine programs and procedures and they must be fluent in an applications language to implement useful non-systems related problems.

The introduction of large time-sharing systems in the early 1960's stifled the development of high-level programming environments. Since one of the primary goals of such systems was to provide an environment supporting a number of programming languages and applications, a low-level command language (*i.e.*, the operating system interface) became the common denominator through which users of different languages could interact with the system. With the advent of high-speed personal workstations, the dependence on time-sharing systems has significantly diminished. Despite this trend, most workstations still maintain the legacy of their timesharing ancestors by continuing to enforce the separation between command and application languages.

Herring and Klint[41] write that three necessary conditions that need to be satisfied in designing a language-based computer system are: (1) elimination of any distinction between programs and procedures (2) support for implicit persistence – no distinction should be made between the naming and typing of files and variables and (3) directories and libraries and other modularity structures found at the command-level should be available as data structures in the language. In other words, users (and application programs) should have free access to the structure of the system environment. Few languages fill any of these requirements, however; Smalltalk[38], Interlisp[62] and related Lisp Machine environments, and Cedar[61] are perhaps the most no-

[5]A notable exception to this is the JOHNNIAC system developed by the Rand Corporation[44] which was perhaps the first proposed design for a monolingual programming environment.

table exceptions. These systems present an integrated programming environment to the user; they facilitate user access and manipulation of system-maintained structures by combining the requirements of both the command and applications language into a single, unified framework. Almost all interaction between the user and system can be mediated through the same language used to express regular applications.

The Symmetric Lisp programming environment is similar in spirit to other monolingual environments insofar as it attempts to eliminate any distinction between system-routines and user programs: users can extend, restructure and customize the system to their choosing since all operations on system-maintained structures are written in the same language as application programs. This is accomplished, not by adding extra data types and structures to accomodate the special needs of a command-level language, but by couching all command-level operations in terms of maps and map access. Our discussion in the following sections focuses primarily on linguistic issues related to programming environments. There are a number of important questions not pertaining to the semantics of the command language that are, nonetheless, very relevant to the design of a usable programming environment (*e.g.,* file protection, resiliency, scheduling policy among users, etc.). A truly workable Symmetric Lisp programming environment must address these issues in detail; the *implementation* of protection, persistence, etc. in the context of Symmetric Lisp is an important topic for future research.

5.2.1 Parallelism

Unlike any of the other language-based systems cited above, however, a Symmetric Lisp-based system is intrinsically parallel. Once an expression has been read, its evaluation can proceed with expressions input subsequently. Because the front-end defines a map object, input expressions that refer to names whose values are not yet known simply block (as per the normal name evaluation rule) until a value is computed, and then proceed as usual.

What use is parallelism in this context? Users often need to perform several computations simultaneously. Conventional machines use multiprogramming or logical concurrency to support parallelism; parallel machines can support true concurrency. The important question is how concurrency is presented to the user.

Conventional solutions along the lines of a Unix fork are not very satisfactory because they place on the file system the burden of keeping track of the results yielded by the background processes.

Users can't exploit this parallelism very effectively because each background process executes independently of the others; sharing or communicating partial results is completely under the control of the file system. Moreover, each process executes in a separate address space. Process forking is also expensive in such systems because of the cost of a context switch. Alternative solutions like that used in Lisp-based machines[39] allow users to express parallelism explicitly at the interpreter-level, but only by wrapping expressions that are to be evaluated concurrently inside a special **process** construct. Lisp processes are, themselves, implemented using expensive coroutines; all processes execute within a flat namespace and, like a Unix-generated process, can't be encapsulated within other data structures. They are not integrated into the rest of the language because their semantics remain very much implementation dependent. To date, there are few commerically-available parallel language-based workstations[6]; user interfaces for most existing parallel machines are usually implemented using Unix or some similar variant.

Symmetric Lisp, on the other hand, allows concurrent evaluation to coalesce naturally into a shared naming environment. Suppose, for example, that a user chooses to run four test cases of some function in parallel. He might type:

```
test-cases : (map
                  first  :  (f first-args)
                  second :  (f second-args)
                  third  :  (f third-args)
                  fourth :  (f fourth-args))
```

The value returned by the first test-case will be accessible, as soon as it completes, under the name **first**; to inspect this value, the user simply types:

```
test-cases.first
```

to the evaluator. He might type **test-cases.first** even before the computation is complete. Evaluation of the expression **first** accordingly blocks – but the front-end, because of its non-strict behaviour (*i.e.*, because of the semantics of map evaluation), doesn't block: it stands ready to accept new input despite the fact that previous expressions, in this case '**first**'' is still under evaluation. If, after generating the four parallel test-case jobs, the user chooses to format the results of the first two and send them to the printer, he might enter:

```
(print (format test.first test.second))
```

format would block until "**first**" and "**second**" become available.

[6]The Cogent-XTM system running Linda and the Xerox Firefly system running Modula-2 are notable exceptions.

How is functionality of this sort different from that found in Unix or modern window-based command monitors? Unlike these systems, the Symmetric Lisp evaluator operates without t[illegible] extraneous influence of a file system or a window monitor. It interfaces cleanly to other pie[illegible] of an expanding program structure because the environments that it manipulates are directly accessible to the user.

Functionality of this sort can probably be expressed in other non-strict, interpretative languages (*e.g.*, parallel Lisps or interpreted non-strict functional languages) by encapsulating every input expression inside of a process creator (*e.g.*, a MultiLisp future[40]). But, note that such a solution places the burden of maintaining the environment structure on the front-end; parallelism is managed explicitly by the top-level evaluator, not the environment structure that is being augmented. Because the front-end must manage the interaction of processes, it is responsible for implementing the scope rule implicit in the semantics of open-maps: element k cannot begin evaluation until the name to be defined by element $k-1$ has been installed in the environment.

Since one can input **map** expressions at the top-level, one can also build local parallel threads of computation in the midst of an interpreter session. For example, to evaluate a **cobegin-coend** statement, one need only input a **map** containing an element corresponding to each element in the parallel form. Consider a parallel algorithm that is structured as a controller and a series of identical workers. Entering the expression:

```
    (map (control) (worker) (worker) (worker))
```

creates a four-thread parallel computation; as usual, the evaluation of this expression yields another well-defined object that can be examined and selected but does not cause the front-end to block waiting for the result.

5.2.2 File-Systems and Naming Environments

Persistency refers to a data structure's outliving the program that created it. Few programming languages support persistency; most require instead the use of an external storage agent such as a file-system to manage long-lived data. Consider the three monolingual systems mentioned earlier: Interlisp retains the distinction between files and other data objects; files cannot be defined using the language's standard type-definition facility. Files are managed instead by a separate *file-package manager that is responsible for maintaining* all *permanent* libraries. Like Interlisp, Cedar distinguishes between long-lived and temporary data. Users have the responsibility of interacting with the file-system whenever data is to be made permanent; they cannot specify that a structure is to be permanent without first writing it into a file. Unlike InterLisp, all permanent data in Cedar is represented in terms of file types.

Smalltalk differs from both InterLisp and Cedar in its support for persistency: all class definitions in Smalltalk are assumed permanent; local definitions and non-class objects are not. Although one might consider the Smalltalk class to be the analogue of a Cedar file, there are important semantic differences between the two. A Smalltalk class need not be enveloped inside a file-object to become persistent – all classes are, by definition, persistent objects. A Cedar file, on the other hand, is a permanent object but the only operations allowed on it are primitive read and writes – its semantics are independent of the semantics of the structures which it encloses.

Environments in Symmetric Lisp are potentially persistent objects – a map's lifetime is independent of the expression that creates it. Any element attached to an interpreter-session open-map exists for as long as the open-map does (unless the user explicitly removes it). Such an environment can be made one element of a global "world-map" that is assumed to exist indefinitely. This view of persistence is similar to that found in PS-Algol[10, 11].

The global world-map is built when the system is initialized. It remains extant forever and is never discarded. Here again, we note that we are not addressing the means by which the world-map becomes persistent; our focus in this section is to discuss how, given a means by which persistent objects may be built, first-class environments may be used to implement applications common in modern-day file-systems.

Persistence is a transitive property – any object attached to the persistent world-map is also persistent. Since there is no constraint on what objects may be attached to an open-map, any object can become potentially persistent. Users need not explicitly state which objects are to be made persistent and which are not; if the front-end, upon close of a session, attaches the newly-created environment object to the world-map, every element in this interpreter-session map will outlive the front-end activation. Thus, if the result of an interpreter-session is found in **my-env-today**, the expression:

```
(layer (map today : my-env-today) world-map.my-user-env)
```

records the session as an element in **my-user-env**; this environment is part of the persistent **world-map** and can be referred to by evaluating

```
world-map.my-user-env.today
```

5.2.2.1 The File System

Assuming the provision of a top-level persistent map, one can use environments as file systems. They have the attributes that are important in modern file systems: they are named collection of named elements that may be freely nested. Thus, a file system might take the form:

```
file-system : (map
                  lock-file-system : lock
                  directory1 : (map ... )
                  directory2 : (map ... )
                     ...
                  directoryn : (map ... )... )
```

Such a file system has the unusual characteristic of having a structure, type and organization that is completely specified by the user. Users are free to restructure the file-system as they choose because the organization of the system and the operations defined over it can be fully expressed in terms of Symmetric Lisp maps and map-operations. In this sense, one can view Symmetric Lisp simply as a base-language that provides the mechanisms upon which to implement more sophisticated file system policy decisions. The file-system would be an element of the global world-map.

To add a new directory to the **file-system**, one might evaluate:

```
(holding file-system.lock-file-system
      (set file-system
              (layer file-system (map  directoryn+1 : < new-directory>))))
```

This expression atomically adds a new layer containing *directory*$_{n+1}$ to the file-system map; by seizing the **lock-file-system** lock before performing the update, we guarantee that the append operation is atomic.

A directory is structurally similar to a file-system and is defined by the following directory template:

```
make-directory : (kappa (protection-info user-group number-of-files-allowed)
                     lock-directory : lock
                     files : (map)
                     get-file : (lambda (... ) ...)
                     size-info : (lambda (... ) ...)
                     delete-versions : (lambda (... ) ...)
                         ... )
```

One creates a new directory by evaluating `(make-directory)` giving the appropriate arguments. In the above formulation, **files** is accessible to all users of the directory. It may be desirable, however, to restrict free access to the **files** field, for example, in situations where the system needs to make sure that a user has the required access rights to the directory and file before allowing him to examine the file's contents. One can implement a simple restricted access policy by declaring the **files** map to be private. Users of the directory would no longer be able to examine the contents of the **files** map *by name*[7] directly, but the procedures declared within the **kappa** – **get-file**, **size-info**, etc. – do have access to the file map because they are found within the same lexical environment. Thus, in a protection-based implementation of a directory system, one would access a file in a directory `dir` by evaluating (`dir.get-file` *file-name*) where *file-name* would presumably be the string representation of the file. (The means by which strings are converted to names is described below.) Note that any reasonable implementation of the language would ensure that the body of all function definitions found within the kappa are shared among the kappa's different instantiations.

A file is any Symmetric Lisp object; one can add a new file to a directory in the same way that one adds a new directory to a file-system:

```
(holding directory.lock-directory.
   (set directory.files
        (layer (map  new-file) directory.files)))
```

In order to manipulate files and directories, it is important to be able to operate over names and bindings flexibly. To this end, the language contains several coercion operators between strings and maps. The expression:

(**make-map** s_1 e_1 s_2 e_2 ... s_k e_k)

($k \geq 1$) builds a k element map. Each of the s_i is either a string (containing alphanumeric characters, not including space or return) or the keyword **nil**. Each of the e_i is an expression. The **make-map** operator returns a map expression:

(**map**
  n_1 : e_1
  n_2 : e_2
  ...
  n_k : e_k)

[7]Unfortunately, they would still be able to access the elements of a directory by position. To prevent this situation we must extend the semantics of maps to allow the user to prohibit position-based selection of a map's elements if he so desires.

where the n_i denote the name representation of string s_i; if s_i is **nil**, then the i^{th} element in the created map expression is simply e_i. Each of the e_i evaluate in the evaluation environment of the **make-map** expression. **Make-map** is useful when one wishes to build a map given a string representation of a name. For example, when adding a new file to a directory, the file-name will presumably be represented as a string; using **make-map**, one can convert the string to a Symmetric Lisp name. If the string bound to identifier **file-name** is "**foo**", the expression

```
(holding directory.lock-directory
   (set directory
        (layer (make-map file-name < contents>) directory.files)))
```

adds the binding **foo** :<*contents*> to **directory.files**; if there already existed a file named **foo**, *contents* supersedes **foo**'s old binding-value.

Most modern file-systems support some form of file version management: users can place several files with the same name in the same directory; a version number is usually attached to each file to disambiguate it from other files with the same name. The file-system sketched above does not support version management – because of the semantics of **layer**, maps containing files with the same name will simply supersede old definitions with the new ones. Once superseded, it would be no longer possible to retrieve the old file.

To support version management, some facility is needed to operate over names (or their string equivalents). In order to manipulate names as strings, the language also provides an operator that behaves like the inverse of the **make-map** constructor; the expression, (**names-in-map** M), returns a map containing the string representation of all names defined in M. Thus, given the directory:

```
M : (map
         protection-info : ...
         user-group : ...
         number-of-files-allowed : ...
         lock-directory : ...
         files : (map
                     f1 : ...
                     f2 : ...
                        ...
                     fn : ... ))
```

the expression, (**names-in-map** M.**files**) returns the map

```
(map "f1" "f2" ... "fn")
```

Every binding name in `files` is represented as a string and returned in a map; note that given the semantics of maps, there cannot be duplicate file names in *M*.`files`.

Given string operators like those found in Common Lisp and the ability to convert Symmetric Lisp names into a string representation, one can proceed to build a simple version management system. For example, suppose that the version number of a file is attached to the end of the file-name, separated by a period, thus: *file-name.version-number*. One can now use the following function to determine the number of files with the same name in a given directory:

```
get-current-version: (lambda (directory file)
                        file-names : (names-in-map directory.files)
                        (map-reduce +
                          (onto (lambda (f)
                                  (if (string= (string-right-trim "." f)
                                               file) 1 0))
                                file-names) 0))
```

The `map-reduce` function behaves like the reduce operator found in Common Lisp except that it operates over maps (not lists); the `string-right-trim` function strips off all characters in `f` starting from the right up-to and including the period; this operation removes the version number information from all files found in the directory. The `onto` operator simply records a 1 for every file-name that matches the input argument, and a 0 for every one that doesn't. The elements of the returned map are then accumulated by the `map-reduce` operation.

Once given a means of determining the current-version number of a file, one can now write a routine to add a new file to a directory with the appropriate version information attached to it:

```
add-file : (lambda (directory file-name contents)
             current-version : (get-current-version directory file-name)
             new-version-number : (1+ current-version)
             new-file-name : (concatenate file-name "."
                                (prin1-to-string new-version-number))
             (holding directory.lock-directory
                (set directory.files
                     (layer directory.files
                        (make-map new-file-name contents)))))
```

Thus, if *M*.`files` is the map,

```
(map
  my-file.1 : < contents-1>
  my-file.2 : < contents-2>)
```

the result of evaluating (`add-file` *M* `"my-file"` <*new-contents*>) causes *M*.`files` to become

```
(map
  my-file.1 : < contents-1>
  my-file.2 : < contents-2>
  my-file.3 : < new-contents>)
```

One can build upon this simple framework to express more complicated version management operations. For example, one can define a function that returns the contents of the i^{th} version of a file as follows

```
string-to-exp : (lambda (string-exp)
                   read-stream : (open-map.create)
                   (attach read-stream string-exp)
                   (read read-stream))
get-contents : (lambda (directory file i)
                  file-name : (concatentate file "."
                                  (prin1-to-string i))
                  (mlast
                     directory.files.(string-to-exp file-name)))
```

`file-name` is bound to the string representation of the file name whose contents are needed; the `string-to-exp` function converts the string into an expression via the `read` operator. The object returned by `string-to-exp` is a map whose evaluation environment is `string-to-exp`'s dynamic environment; in this case, it will be the environment defined by `directory.files`.

An unwanted consequence of this solution is that the search for the file will continue outward to the evaluation environment of `get-contents` if the file is not found in the directory[8]. Despite this deficiency, this solution nonetheless illustrates how the built-in name-lookup semantics and the use of a layering operation could be used to implement non-trivial directory management routines; outside of the two coercion operators to build strings from names, and to build maps from strings, no other extensions to the kernel language were required. The file-system given here is by no means the only one implementable in the language; users are free to restructure their file system or to implement other version management procedures if they wish.

Consider another example. To delete all versions of a file, one need only seize the directory and build a new map containing all the current file entries in the directory except for the file versions to be deleted. Because map elements are shared, the overhead of this procedure is only in the copying of bindings; the contents of the files themselves are not copied.

[8]An exception handling mechanism could be used to prevent such a situation from occuring; the linguistic issues relating to an exception-handling facility is not difficult to understand; a simple non-traversing variant of `with` can be used to express a primitive exception facility. On the other hand, the implementation of such a mechanism in the context of a non-strict evaluation semantics is a more problematic question.

```
delete : (lambda (directory file-name)
   new-file-structure : (map-reduce layer
                           (map-remove nil
                              (onto (lambda (f)
                                       (if (string= f
                                              (string-right-trim ''.'' file-name))
                                           nil
                                          (make-map f directory.files.f)))
                                 (names-in-map directory.files))) (map))
   (holding directory.lock-directory
       (set directory.files new-file-structure)))
```

Evaluating the `onto` expression yields a map whose elements are either `nil` or a one element map containing the binding associating a file-name with the file's contents. The `map-remove` function returns a new map in which `nil` elements have been removed; the `map-reduce` operator performs a union (using the `layer` operator) of the sub-maps found in the map returned by `map-remove`. One can easily rewrite this function to remove a single version of a file if one chooses.

The version management system described above is by no means the only one possible. An alternative approach would have been to treat a file as a map of versions. In this scheme, the i^{th} element in the file-map would represent the i^{th} version of the file. Thus, the contents of M.`files` shown above may be structured as follows:

```
files : (map
           my-file : (map
                        contents of my-file.1
                        contents of my-file.2))
```

In this scheme, the `add-file` function would be rewritten as follows:

```
add-file : (lambda (directory file contents)
              (holding directory.lock-directory
                 (set file (layer file (map contents)))))
```

This function assumes, as did `get-contents`, that the directory contains the file in question. Unlike the `add-file` function used to add files in directories with explicit version numbers, the arguments to the above function are not strings; if we wished to add a new version to a file named `f` in directory `d`, we evaluate, (`add-file d f` <*contents*>). The function simply appends the *contents* onto the end of `f`; it locks the directory to prevent other users from accessing `f` while the append takes place. In this scheme, the `get-current-version` function would be replaced by an `mlast` operation:

```
(get-current-version directory file) ≡
```

```
(mlast directory file)
```

where, once again, **file** is not a string representation of the file name but rather the contents of the **file** itself.

Note that in the versions-as-separate-maps scheme, versions are treated files; the version number associated with a given file is fixed at the time the file is created. Thus, a file with name **my-file.3** retains this name even if **my-file.2** is subsequently deleted. In the versions-as-regions scheme, versions are not treated as separate files; the definition of the "i^{th} version of a file" changes if previous versions of the file are subsequently deleted. Symmetric Lisp supports both kinds of version management equally well.

Another important file-system command that is easily implemented using maps and the standard string operators is file-name completion. Given a string, s, and a directory, one requires a procedure that returns all file-names in the directory of which s is a prefix.

```
file-complete : (lambda (directory substring)
                  files-names : (names-in-map directory.files)
                  (map-remove nil
                     (onto (lambda (file)
                             (if (string< substring file)
                               file
                               nil))
                      file-names)))
```

file-names is bound to the string representation of all file names found in the directory. The onto expression performs a string comparison on each file, returning the string representation of the file-name if the input string is a prefix of that file.

How is the structure of the file-system given here different from one that would have been devised in other high-level, expression-oriented languages? The feature of Symmetric Lisp that clearly dominates the design of the file-system is the language's support for first-class environments. Name-lookup and namespace management are fundamental tasks of any file-system; a language-based implementation of an operating system in which the base language does not support environments as first-class data types means that the implementor of the file-system must explicitly provide routines to handle namespace management. Because these operations are primitive tasks in Symmetric Lisp, the Symmetric Lisp programmer has freedom to structure and access the file-system in any way he chooses.

On the other hand, it may be argued that the truly important issues in the construction of a language-based system are not those that deal with the *linguistic primitives* that the base language ought to provide but rather those that address the *implementation* of persistency and protection. It is certainly possible, for example, to implement a file system based on the structure we have here in a language like Common Lisp using, for example, alists rather environments as the fundamental structuring tool. Our claim is that first-class naming environments provide a more satisfactory basis upon to which to build a file-system than structures like alists. On the other hand, it is clear that this is not the whole picture; a viable base language must be accompanied by an efficient implementation of persistence and protection; we come back to this issue in Chapter 8.

5.2.3 Meta-Cleanliness

Parallelism and persistence in Symmetric Lisp combine to form an interesting and vigorous symbiosis. Because all file-like operations in Symmetric Lisp are actually operations over maps, parallelism is inherent in any file operation. The ability to treat maps as long-lived, lightweight processes coupled with their role as first-class environments leads to other kinds of expressiveness not easily available in other language-based systems.

As a first example, consider the following situation: the user wishes to run many test cases of a program `Q` concurrently; he wants to analyze the results of each run using an analysis program, **`analyze`**; whenever an analysis turns up a "good result", he wants the result entered into a directory named **`good-runs`**.

One convenient way to go about this is to create a new environment called **`test-runs`** that contains the test runs desired:

```
test-runs : (map
               (Q < first-test-args>)
               (Q < second-test-args>)
                  ... )
```

All test runs will evaluate in parallel. One can now set up the analysis state by evaluating `(onto analyze test-runs)`. The result of this expression is a map containing the result of applying **`analyze`** to each test-run; if this expression is bound to **`analyze-cases`**, one can determine the "good" runs by evaluating

```
(map
   good-runs : (make-directory < protection-info user-group number-of-files-allowed>)
```

```
    (onto (lambda (analyzed-case)
                (if (check-out analyzed-case)
                    (add-file good-runs ''test-runs'' analyzed-case)))
       analyzed-cases))
```

Each test-run added to the directory will automatically have a unique version number associated with its file name. This is a simple solution to what would be, in most systems, a difficult problem.

The absence of any distinction between application and command levels in a Symmetric Lisp system has other beneficial consequences. One can, for example, store a collection of libraries in a directory and use them directly in application programs without having access to them mediated by system level routines. One might store commonly used functions in a library-directory:

```
library-directory : (map
                        structures-library : (map
                                                 stack : (kappa ... )
                                                 queue : (kappa ... )
                                                     ...
                                                 tree  : (kappa ... ))
                        transcendental-library : (map
                                                     sin : (lambda ... )
                                                     cos : (lambda ... )
                                                         ... )
                        ... )
```

To evaluate a program, *p*, using the functions declared within this library one would evaluate

```
(with (layer library-directory.structures-library
             library-directory.transcendental-library
                 ... )
      p)
```

Assuming that all functions are defined in only one library, all functions declared within all libraries are then accessible to *p*.

One can effect the behaviour of a `load` function commonly found in Lisp environments by evaluating:

```
load-library-env : (layer library-directory.structures-library
                          library-directory.transcendental-library
                              ... )
```

`load-library-env` is bound to an map containing the most recent definitions of all routines defined in the various libraries contained in the library directory. To set up an interpreter-session in which all expressions evaluate in the environment specified by `load-library-env`, one would evaluate:

```
(FE load-library-env io-stream)
```

Because all definitions are recorded as part of the evaluation environment of the open-map, free names referenced by expressions input to this `FE` are first resolved by searching through `load-library-env`.

5.2.3.1 Creating Daemon Processes

A daemon is a passive process that watches a data or program structure waiting for new developments. Background jobs in conventional multiprogrammed systems can be used to build daemon-like processes, but such jobs have limited applicability in these systems because they are built and managed by command-level, not applications-level, programs. In Symmetric Lisp, on the other hand, it is easy for the user to create daemon processes because there is no boundary separating the structures manipulated by the user and those manipulated by the system. Some examples are given below.

Example 1:

Consider the following definition of a mail-daemon:

```
mail-daemon : (lambda (mail-stream)
                 new-mail : (mcar mail-stream)
                 (seqmap
                    (display-msg new-mail)
                    (mail-daemon (mcdr mail-stream))))
```

To use the daemon, a user types (`mail-daemon my-mail-stream`) to the front-end. The evaluation of this function runs indefinitely – but, the non-strict semantics of maps implies that the front-end itself doesn't hang; it is ready to receive new input despite the fact the evaluation of a previous expression, here the function application, is still ongoing. Whenever a new message is appended (using attach) to `my-mail-stream`, the daemon displays the message and then quiesces to await the next one. Users can simultaneously read mail by traversing `my-mail-stream` using `mcar` and `mcdr`.

Example 2:

Consider a function that is to print a message whenever a user attempts to redefine a keyword during his interpreter session. Such a function can be conveniently written as a daemon process that watches the environment stream constructed by the front-end.

```
redefine? : (lambda (user-env keyword-map)
                 new-item : (get-name (mcar user-env))
                 (seqmap
                    (if (map-reduce or
                            (onto (lambda (keyword-item)
                                       (string= new-item keyword-item)) keyword-map))
                       (write (format "Attempt to redefine keyword:  S",new-item)))
                    (redefine? (mcdr user-env keyword-map))))
```

This daemon checks if any element added to **user-env** defines a region using a keyword. **Keyword-map** is a map containing the string representations of names the user wishes to be reserved. The **get-name** special-form is a restricted version of **names-in-map**; it evaluates its first argument to get a map-region (*i.e.*, an l-value) and returns the string representation of the name associated with that region if one exists, or returns **nil** otherwise. In the above example, **new-item** is bound to the string representation of the region-name of the current head of **user-env**.

It is interesting to note that this daemon can execute from within the open-map that it is examining. That is, if front-end E is building environment **user-env**, the user can input:

```
(redefine? user-env keyword-map)
```

to E. Because maps are distributed data structures, **redefine?**'s activation can access **user-env** even though it is strictly a component of that open-map. Because maps are non-strict, the front-end can proceed to accept new-input while **redefine?** continues to execute. If we assume that E builds **user-env** by side-effect, **redefine?**'s activation will be able to examine new input as they become available.

5.3 Summary

The primary focus in this chapter has been on describing how one might construct a monolingual programming environment implemented in Symmetric Lisp. There are three prominent features of the language make it a good candidate as a base language for a monolingual computer system:

1. Environment management is a fundamental part of the language's semantics. This means that the Symmetric Lisp front-end need not maintain any internal environment image:

it repeatedly executes a **read-layer** operation rather than executing the conventional **read-eval-print** loop. Expressions input to the front-end execute within, and resolve into, a unified name-space.

2. Parallelism is fundamental in the evaluation of every map expression. This means that the front-end is a parallel program: expressions input to it are executed concurrently with other input with the only constraint being that imposed by the name-evaluation rule. Parallelism also implies that users are free to build daemon processes to monitor the evolution of running programs or to watch communication streams for developments.

3. We require the existence of a persistent world-map structure to which any Symmetric Lisp object may be attached. Because maps are the fundamental program as well as data object in the language, any data object, not just files or directories, can be stored on the persistent image. This includes the environment built by the front-end during the course of an interpreter session.

In the introduction, we had observed that the symmetric model supports *uniformity between language-level and system-level.* We envision Symmetric Lisp as a high-level operating-systems language or as a base language in a heterogenous environment. Because it provides a good foundation for building flexible system-level structures, we believe that the same techniques used to show the relationships between superficially diverse program structures can be used to also relate superfically diverse structures found in *different* domains; for example, the structure of the object yielded at the end of a session with the Symmetric Lisp front-end possesses the same structure (and is built in the same way) as the objects manipulated in a Symmetric Lisp-based inheritance system.

The following chapters address issues of implementation. The next chapter focusses on compilation issues, concentrating in particular on the question of determining the proper evaluation environment of identifiers. Chapter 7 discusses a source-to-source transformation of Symmetric Lisp expressions into a dataflow language.

Chapter 6

Name-Translation

First-class environments raise many interesting compilation-related issues. The ability to parameterize and extend tne evaluation environment of expressions, the uniformity of program and data structures, and the fine-grained parallel evaluation semantics of maps makes the Symmetric Lisp compiler's task a challenging one. In this chapter, we examine the issue of name-translation in the Symmetric Lisp context: how can we statically infer the proper evaluation environment of any identifier found in a Symmetric Lisp program?

Unlike block-structured or functional languages in which the lexical environment of the reference defines the variable's evaluation environment, the presence of scope-expressions in Symmetric Lisp means that there may be several possible evaluation environments for a given identifier. Because of the language's ability to parameterize the evaluation environment of expressions, the issues that arise in inferring the proper evaluation environment of Symmetric Lisp identifiers are quite similar to those that arise in inferring the proper method and class definitions for identifiers in late-binding languages like Smalltalk[60].

The information yielded by a procedure that determines the evaluation environment of identifiers in a Symmetric Lisp program would be of greatest use to the Symmetric Lisp code-generator. A code-generator could use the information provided by this procedure to translate symbolic name references into a more more efficient base-language representation. In the absence of such a procedure, the evaluation environment of identifiers found in scope-expressions would need to be determined dynamically by the runtime kernel. Most implementations of late-binding languages (*e.g.*, Smalltalk[27]) usually require sophisticated runtime name-translation facilities to achieve reasonable execution performance. The ability to statically determine the environment containing the ap| ·priate binding for an identifier would relieve the runtime kernel of the burden of performing such a task; the compiler can directly generate code that accesses an identifier's binding value from the base-language environment structure provided that it knows the environment in which to look.

In this respect, the motivation for implementing a name-translation mechanism is in many ways similar in spirit to the motivation underlying strictness analysis mechanisms[21] used in the compilation of lazy functional languages. In an implementation of a lazy functional language employing strictness analysis, the compiler tries to determine which arguments to a function can be evaluated in parallel with the application of the function by evaluating the strictness of the function with respect to that argument. The parallel evaluation of strict arguments and function application increases the concurrency of the program but does not compromise the language's normal-order semantics. Our approach to name-translation serves a similar purpose: by mapping all expressions to a greatly simplified domain that gives information about the names they define, we can improve execution performance (since runtime name-translation is reduced), while still preserving program meaning (as specified by the language's operational semantics).

The problem of determining the evaluation environment of Symmetric Lisp identifiers is intimately related to the problem of determining the *structure* (or *type*) of environment-yielding expressions. For example, consider the expression:

```
f : (lambda (x)
      (with x id))
```

The meaning of the free variable `id` in the scope-expression is dependent upon the structure of `x`: if *all* applications of `f` in the program define a binding for `id`, then the evaluation environment of `id` can be determined to be the environment yielded by evaluation of `x`. On the other hand, if it can be determined that `f` is *never* applied to an actual that defines `id`, then the value of `id` in `f` should be determined by its binding value in `f`'s lexical environment. In both of these cases, the reference to `id` can be efficiently translated: in the latter case, one can translate the reference to be the address of `id` in the lexical environment; in the former case, the reference can be translated into a lookup-operation into the environment structure denoted by `x`. It is possible based on the semantics of `with`, however, that `x` *inconsistently* defines `id`; consider the following two calls to `f`:

(f (map id : $< exp >$))

```
(f (map))
```

The first application defines a binding for **x** whereas the second one does not: in the first application, the value of **id** should be its binding-value in **x**; in the second application, its value should be retrieved from **f**'s lexical environment. In this scenario, there is no efficient translation of **id** possible – a runtime search on the environment activation structure is necessary in order to determine the environment in which it is defined.

As a second example, consider the function:

```
g : (lambda (x)
        (1+ (mlast (select x y))))
```

The operational semantics of **select** tells us that **y** *must* be defined in **x**. The free reference to **y** can be compiled into a lookup-operation on the environment defined by **x**; no dynamic search along a runtime environment structure is necessary in this case.

As a third (and related) case, we can assert that the map object yielded by evaluation of the expression:

```
h : (map x:1 y:2)
```

never defines any other name besides **x** and **y**[1]; thus, if this expression is ever used as an environment-specifier in a scope-expression, *e.g.*, (**with h (+ x z)**), examination of **h**'s structure tells us that the reference to **x** should be resolved using its binding-value in **h** whereas the reference to **z** should be resolved based on its binding-value in the lexical environment of the scope-expression. Similarly, one could also assert that the map object yielded by evaluation of the **select** expression in function **g** never defines any names besides **x**.

The central observation that should be made from the above examples is that the *type* of an environment-yielding expression must say something about the names defined by that expression. The type of a map expression, for example, should, in addition to specifying the types of its component regions, also indicate that the map *always* defines the names bound to its regions. On the other hand, the type of an environment-specifier of a scope-expression may not always define the same set of names (*e.g.*, consider identifier **x** used in function **f** above); its type should indicate which names it always defines and which names it sometimes (or inconsistently) defines. For example, one possible type that can be inferred for **g** is a function type

[1]Assuming, of course, that **h** is not side-effected to a map-object that defines other names.

that maps from environments defining name `y` to integers. Similarly, one possible type for `h` is an environment type that maps `x` to integers and `y` to integers.

It is instructive to determine whether the name-translation problem is indeed made any easier in the absence of scope-expressions. Let SL^{-SE} be a subset of Symmetric Lisp that does not support scope-expressions. In the absence of scope-expressions, the map structure that defines an identifier's lexical environment is the only evaluation environment of interest[2]. This assertion can be stated formally as follows:

Proposition 6.1 *Let M be a Symmetric Lisp program containing no scope-expressions. Then, the evaluation environment of an identifier in M is always its lexical environment.*

Map structures in SL^{-SE} cannot affect the evaluation environment of other expressions; while they may be projected (via **select**) or selected by position (via **index**), they cannot be used to change the evaluation environment of any other expression as is possible via **with**. This is not to imply that maps may not be passed as arguments to (and returned as results from) functions; what the proposition does imply is that the bindings defined by a map expression are accessible only to other expressions defined within the map. One can access the values a map-object defines by position – this ability is orthogonal to the issue of name-lookup, however, and requires no sophisticated compilation support to implement. Insofar as the name-translation problem is concerned, Symmetric Lisp programs that do not contain scope-expressions can be compiled through standard techniques used to implement other lexically-scoped languages.

This chapter presents an extended type inference system (called an *environment inferencing* system) which can be used as the basis for implementing a procedure to infer the proper evaluation environment of identifiers in a restricted subset of the language; the subset includes higher-order functions, projection and selection operators, scope-expressions and conditionals, but does not contain assignment or **fill** expressions. We discuss the interaction of assignment and environment inferencing in Section 6.1.6.

6.1 Environment Inferencing

An environment inferencing system is an extended type inference system that can be used to deduce information about the names defined by an expression. As discussed above, the relevant

[2]For the purposes of this discussion, we ignore the presence of the **apply-env** operator.

question is how to statically determine the evaluation environment of an identifier found in the body of a scope-expression. Why should we assume that the problem is even a tractable one? One distinguishing property of the subset of Symmetric Lisp of interest to us here is the fact that the core language[3] does not allow names to be generated dynamically – there is no such thing in the language as a "name-yielding expression". This restriction means that the compiler knows every name defined by the program and, moreover, the compiler also knows all the names defined by any given map. (In the full language, operations like **read** destroy this pleasant property – the object returned by **read** may include names not defined elsewhere; **read**, in other words, is indeed a name-yielding operator.)

The inference system defines a collection of *inference rules* that relate expressions to types. The type of an expression gives information about the structure of the object defined by that expression; in the type domain of interest to us here, environment-yielding expressions will have a type that will give information about the types of its region as well as information about the names it defines. In the next two sections, we define the syntax and semantics of types in our system; Section 6.1.2 describes the notion of substitution used to capture polymorphism in our type rules and Section 6.1.3 discuss the subtype relation ordering on types. Section 6.1.4 defines the inference rules.

6.1.1 Syntax

6.1.1.1 The Expression Language

Assuming a set of identifiers x, the language SL_O of expressions e of interest is given by the following grammar:

Terminal and keyword symbols are shown in `type-writer` font, and non-terminals are displayed in roman font. Alternatives are separated by double vertical bars (|). The form $\{Exp\}^*$ indicates that Exp may be repeated zero or more times; the form $< Exp >$ indicates that Exp is an optional expression:

```
Expr        ::= Integer and Boolean Constants
            ::= Id
            ::= (lambda (Id) Expr)
```

[3] For the present discussion, we consider the language as defined in Chapter 2 to constitute the "core" language.

::= (Expr Expr)
::= (**select** Expr Id)
::= (**with** Expr Expr)
::= Expr[Expr]
::= (**if** Expr Expr <Region>)
::= Arithmetic Operations, Boolean Operations
::= (**map** {Id : Expr}*)

The subset of Symmetric Lisp used in this chapter consists of maps, name and region selector operations, lambda abstraction and application. In addition, the subset includes scope-expressions, conditionals as well as some basic arithmetic and boolean testing forms. All user-defined functions are assumed to be single-arity and applications of lambda-defined functions take only a single argument. Note also that we assume each region in a map is named[4].

6.1.1.2 The Type Language

Every expression in the expression language is associated with an element in the type language by the inference system. We present the syntax below and describe the semantics of the type elements following:

Type-Scheme	::= Type | $\forall\alpha$ Type-Scheme
Type	::= Type-Variable | Primitive-Type | Map-Type | Function-Type | $\top_t$ | $\perp_t$
Occurrence	::= **a** | **n** | $\top_o$ | $\perp_o$
Primitive-Type	::= **Integer** | **Boolean**
Type-Variable	::= α and its subscripted and superscripted variants
Map-Type	::= (M,O)
M	::= $M_\perp$ | $M_\top$ | M[Id $\mapsto$ Type]
O	::= $O_{\mathbf{n}}$ | $O_\top$ | O[Id $\mapsto$ Occurrence]
Function-Type	::= Type $\rightarrow$ Type

There are four basic types in our type language: (1) *type variables*, (2) *primitive types*, (3) *map-types*, and (4) *function types.* In addition there are two special type constants: $\top_t$ indicating

[4]This is not a very strong restriction; it is easy to exhibit a transformation that takes a Symmetric Lisp map containing unnamed regions into one that contains only named regions if we simply associate a unique identifier to each unnamed region in the original map.

an "inconsistent" type and $\perp_t$ indicating a "least" type; $\top_t$ is a supertype of all types whereas $\perp_t$ is a subtype of all types.

Type-schemes define *polymorphic* or *generic* types; an expression that has a polymorphic type may be used in a number of different type contexts. A polymorphic type may be *instantiated* to a more refined type by appropriate substitution of type variables for types; a *monotype* is a type containing no type variables. The treatment of polymorphism in our type language is similar to its treatment in the paradigmatic polymorphic language, ML[24, 50]; readers unfamiliar with the ML polymorphic type system are encouraged to review the cited references. Notation: We will often abbreviate a type-scheme

$$\forall\alpha_1\forall\alpha_2\ldots\forall\alpha_n\tau$$

as

$$\forall\alpha_1\alpha_2\ldots\alpha_n\tau$$

Notation: Throughout this chapter, we shall use τ (and its subscripted variants) to range over types and δ (and its subscripted variants) to range over type-schemes.

Environment-yielding expressions are associated with a *map-type*. A map-type is a *pair* of functions, the first component maps Symmetric Lisp identifiers into types, and the second component maps Symmetric Lisp identifiers into *occurrences*. The role of the first component in the pair is to associate every region in a map with a type; the role of the second component is to provide information on whether a given identifier is defined by the map or not. Recall that an identifier defined by an environment-yielding expression may be in one of three contexts: (1) the name is *always* defined by the expression, *e.g.*, we can always assert that a map object with regions named **x** and **y** always defines **x** and **y**, (2) the name is *never* defined by the expression, *e.g.*, we can always assert that a map object with regions named **x** and **y** never defines a region named **z**, or (3) the name is *inconsistently* defined by the expression, *e.g.*, consider the names defined by **x** as defined in the body of function **f** shown earlier. Identifiers which fall into the first case are mapped to occurrence element **a**, those which fall into the second category map to occurrence element **n**, and those that fit the third condition map to occurrence element $\top_o$. The fourth occurrence element $\perp_o$ is used only for technical completness. Because our inference rules will require us to compute joins and meets of occurrences, it is necessary that occurrence elements form a lattice structure; $\perp_o$ is used to complete the lattice. No well-typed environment-yielding expression will ever map an identifier to $\perp_o$. The semantics of map-types is explained in greater detail in Section 6.1.4.1.

Notation: We use σ to range over occurrences and M and O (and their subscripted variants) to represent map-types. If (M,O) is a map type, then $M(x)$ denotes the *type* to which x is mapped by M, and $O(x)$ denotes the *occurrence* to which x is mapped by O. We call M a *type environment* and O an *occurrence environment.*

Functions are associated with a *function-type* that maps over the type domain. Function types and map-types are the only constructors in the type language; they are the only types that are built from other types.

We will explain the role of these types in greater detail below. Before doing so, however, we first explain the notion of substitution which is a crucial part of our polymorphic type system.

6.1.2 Substitutions

A *substitution*, S, is a mapping from type variables to types. If τ is a type, then $S\tau$ is written:

$$\tau[t_1/\alpha_1, \ldots, t_n/\alpha_n]$$

and denotes the type obtained by replacing each free occurrence of α_i in τ with t_i. (We assume that $t_i \neq \alpha_i$.) We say that $S\tau$ is an *instance* of τ. Intuitively, a substitution *refines* a type expression, *i.e.*, a substitution constrains a type expression by removing some type variables.

A type-scheme

$$\delta = \forall \alpha_1 \alpha_2 \ldots \alpha_n \tau_1$$

has a *generic instance*

$$\delta_1 = \forall \alpha'_1 \alpha'_2 \ldots \alpha'_m \tau_2$$

if $\tau_2 = [\tau_i/\alpha_i]$ for some types $\tau', \tau'', \ldots$ and where the α'_i are not free in δ. A type instantiation of this form is written $\delta > \delta_1$. Note that ordinary instantiation acts on *bound* variables whereas generic instantiation acts on *free* variables. It follows that $\delta > \delta_1$ implies $S\delta > S\delta_1$. For example, the type-scheme:

$$\forall \alpha_2 (\textbf{Integer} \rightarrow \alpha_2 \rightarrow (\textbf{Integer} \rightarrow \alpha_2))$$

is a generic instance of the type-scheme:

$$\forall \alpha_1 \alpha_2 (\alpha_1 \rightarrow \alpha_2 \rightarrow (\alpha_1 \rightarrow \alpha_2))$$

Ordinary instantiation eliminates free type variables whereas generic instantiation eliminates bound ones. Because only bound variables are eliminated in a generic instantiation, one can instantiate such a type subsequently with new types. Thus, by appropriate use of quantifiers, it is possible to model general polymorphic behaviour. For example, the polymorphic identity function may be expressed as the type-scheme, $\forall \alpha\ \alpha \rightarrow \alpha$. In this definition, α is a type-variable and the quantification makes the type generic. When we instantiate this type (*e.g.*, consider a substitution of type **Integer** for α), the resulting type is **Integer**. A comprehensive description of polymorphic type systems may be found in [18].

6.1.3 Subtyping

We would like our environment inference system to be as flexible as possible while still prohibiting obvious type violations (*e.g.*, using an integer as a procedure etc.). In thinking about how to structure our type domain to support this goal, we are led to ask how the types of different map structures relate to each other. Consider, for example, the following two map objects:

```
M1 : (map n1 : e₁
          n2 : e₂)
```

and

```
M2 : (map n1 : e₃
          n2 : e₄
          n3 : e₅)
```

It is clearly the case that, provided e_1 and e_3 are of compatible types and e_2 and e_4 are of compatibile type, **M2** can be used in any context that **M1** can[5]. For example, assuming that all the e_i are integers, one can use **M2** in any place that it is appropriate to use **M1** since such contexts will never *require* the presence of **n3**. **M2**'s type, in other words, can be regarded as a *subtype* of **M1**'s type. The importance of ordering map expressions under a subtype rule can be further appreciated when we consider the following example:

```
f : (lambda (M)
```

[5]Actually, we should qualify this assertion: an expression of the form (**with** $< M >$ ... **n3** ...) where M is either **M1** or **M2** may not be type-correct if the use of **n3** is inconsistent with its type in **M2**.

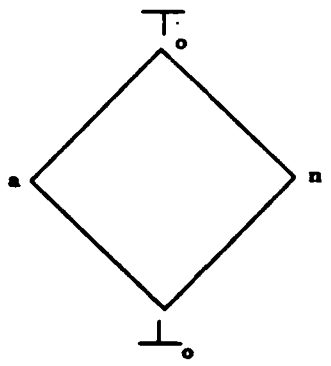

Figure *8* : Structure of the Occurrence Lattice

```
           (select m n))
  g : (f (map n: 2))
  h : (f (map m: 3
              n: 4
              o: 5))
```

f defines a lambda abstraction that projects the name n from its argument map object, m. There are two calls to f in the above fragment: the first applies f to a map that binds n to 2, the second applies f to a map that contains three bindings, one of them being a binding that associates n to 4. It would be certainly correct to consider this fragment as being well-typed since the only constraint imposed by the select operation is that its argument map object (here m) define a binding for n.

Our type system includes a subtyping rule on all types that allows program fragments like that shown above to be considered well-typed. We define the subtype relation as follows:

Definition 6.1 *Let τ_1 and τ_2 be two types. Then,*

1. $\tau_1 \sqsubseteq_t \top_t$ *for any type* τ_1.
2. $\perp_t \sqsubseteq_t \tau_1$ *for any type* τ_1.
3. $\tau_1 \sqsubseteq_t \tau_1$.
4. *If* $\tau_1 = (M_1, O_1)$ *and* $\tau_2 = (M_2, O_2)$, *then* $\tau_1 \sqsubseteq_t \tau_2$ *if for every* $x \in Id$, $M_1(x) \sqsubseteq_t M_2(x)$ *and* $O_1(x) \sqsubseteq_o O_2(x)$ *where* $\sqsubseteq_o$ *is defined by the lattice shown in Figure 8.*
5. *If* $\tau_2 \sqsubseteq_t \tau_1$ *and* $\tau_3 \sqsubseteq_t \tau_4$, *then* $(\tau_1 \rightarrow \tau_3) \sqsubseteq_t (\tau_2 \rightarrow \tau_4)$

The subtyping rule for map-types takes into consideration the value of the occurrence elements associated with the names in the two map-types. If an environment-yielding expression always or sometimes defines a particular name, then the type of the region to which this name is bound is relevant in determining whether it is a supertype of another map-type. Thus, a type environment M_1 is a subtype of M_2 if for each name x possibly defined in M_2, M_1 also possibly defines x, and, moreover, x's type in M_1 is a subtype of x's type in M_2.

Occurrence elements are themselves ordered under a very simple subtype relation: an occurrence element of **a** or **n** is a subtype of occurrence element $\top_o$ and are supertypes of $\bot_o$; moreover, **a** and **n** are themselves incompatibly related under the subtype rule. The intuition behind this is simple: an occurrence element of **a** or **n** imposes a greater constraint on the structure of the associated map object than an occurrence element of $\top_o$; $\top_o$ indicates contradictory or inconsistent information regarding the presence of a name whereas **a** and **n** do not.

The subtype relation rule for function types is interesting because the subtype constraints are inverted for the domain. Intuitively, we would like a function

$$f : \tau_1 \rightarrow \tau_3$$

that is a subtype of a function

$$g : \tau_2 \rightarrow \tau_4$$

to operate in all contexts where g is applicable. This means that f's domain, τ_1, should be less constrained than g's, τ_2 (*i.e.*, $\tau_2 \sqsubseteq_t \tau_1$). Moreover, because f is a subtype of g, we expect f's range, τ_3, to be more constrained than g's, τ_4 (*i.e.*, $\tau_3 \sqsubseteq_t \tau_4$). For example, suppose that f is a function whose domain is a map-type that always defines names n_1 and n_2 and suppose that g is a function type whose range always defines n_3 and n_4. We consider f to be a subtype of g if (a) g's domain defines at least n_1 and n_2 and, (b) f's range defines at least n_3 and n_4. This would allow f to be applied to any object to which g could be applied and would treat the object returned by f to be more constrained than the object returned by g based on the subtype rule for objects. This subtype rule is, in fact, similar to the one used by Cardelli in inferring subtype relations among record objects [16] used in a multiple-inheritance system.

In addition to a subtype relation on types and occurrences, there are two important binary functions used extensively in our inference system. The first is a join function (written $\sqcup_t$) and the second is a meet function (written $\sqcap_t$).

The $\sqcup_t$ of two types is a supertype of these two types; $\tau_1 \sqcup_t \tau_2$ is a type τ_3 defined such that $\tau_1 \sqsubseteq_t \tau_3$ and $\tau_2 \sqsubseteq_t \tau_3$. The $\sqcap_t$ of two types is a subtype of these two types; $\tau_1 \sqcap_t \tau_2$ is a a type

τ_3 defined such that $\tau_3 \sqsubseteq_t \tau_1$ and $\tau_3 \sqsubseteq_t \tau_2$.

We define these two relations formally as follows:

Definition 6.2 *Assume τ (and its subscripted variants) are types. Then,*

1. $\tau_1 \sqcup_t \tau_1 = \tau_1$.
2. $(\top_t \sqcup_t \tau_1) = (\tau_1 \sqcup_t \top_t) = \top_t$.
3. $(\bot_t \sqcup_t \tau_1) = (\tau_1 \sqcup_t \bot_t) = \tau_1$.
4. *If τ_1 and τ_2 are primitive types such that $\tau_1 \neq \tau_2$, then $\tau_1 \sqcup_t \tau_2 = \top_t$. Similarly if τ_1 is a function type and τ_2 is a non-function type, then $\tau_1 \sqcup_t \tau_2 = \top_t$. (A similar rule applies if τ_1 is a map-type and τ_2 is not.)*
5. *If $\tau_1 = (M_1, O_1)$ and $\tau_2 = (M_2, O_2)$, then $\tau_1 \sqcup_t \tau_2 = (M_3, O_3)$ where*
$$M_3(x) = M_1(x) \sqcup_t M_2(x)$$
and
$$O_3(x) = O_1(x) \sqcup_o O_2(x)$$
where $\sqcup_o$ is a join operator defined over occurrence elements in the obvious way given the structure of the lattice shown in Figure 8:
6. $(\tau_1 \rightarrow \tau_2) \sqcup_t (\tau_3 \rightarrow \tau_4) = (\tau_1 \sqcap_t \tau_3) \rightarrow (\tau_2 \sqcup_t \tau_4)$

Definition 6.3 *Assume τ (and its subscripted variants) are types. Then,*

1. $\tau_1 \sqcap_t \tau_1 = \tau_1$.
2. $(\top_t \sqcap_t \tau_1) = (\tau_1 \sqcap_t \top_t) = \tau_1$.
3. $(\bot_t \sqcap_t \tau_1) = (\tau_1 \sqcap_t \bot_t) = \bot_t$.
4. *If τ_1 and τ_2 are primitive types such that $\tau_1 \neq \tau_2$, then $\tau_1 \sqcap_t \tau_2 = \bot_t$. Similarly if τ_1 is a function type and τ_2 is a non-function type, then $\tau_1 \sqcap_t \tau_2 = \bot_t$. (A similar rule applies if τ_1 is a map-type and τ_2 is not.)*
5. *If $\tau_1 = (M_1, O_1)$ and $\tau_2 = (M_2, O_2)$, then $\tau_1 \sqcap_t \tau_2 = (M_3, O_3)$ where*
$$M_3(x) = M_1(x) \sqcap_t M_2(x)$$
and
$$O_3(x) = O_1(x) \sqcap_o O_2(x)$$
where $\sqcap_o$ is a join operator defined over occurrence elements in the obvious way given the structure of the lattice shown in Figure 8:
6. $(\tau_1 \rightarrow \tau_2) \sqcap_t (\tau_3 \rightarrow \tau_4) = (\tau_1 \sqcup_t \tau_3) \rightarrow (\tau_2 \sqcap_t \tau_4)$

If M is a type environment, then $\bigsqcup_t M$ denotes the join of all elements in M's range. A similar operation exists over occurrence environments and meets of type and occurrence environments.

6.1.4 An Inference System

Our heretofore informal description of types and occurrence elements can be made precise by formalizing the notion of a *syntactically well-typed expression.* An expression is considered *well-typed* if a type can be deduced for it based on the rules constituting the type inference system. If no type can be deduced for an expression, we consider the expression ill-typed. A type-system is *semantically sound* if the meaning of any expression as specified by the formal semantics for the expression language is consistent the denotation of the type deduced for that expression by the type-system.

The environment-inference system itself does not define a type-checking algorithm; there will be many possible types that can be deduced for the same expression given different algorithms. Provided that the inference system is sound, these types are all consistent with one another in the sense that they are all refinements of some principal (or most general) type.

The axioms and inference rules for the type system are presented in a form similar to Gentzen's calculus of sequents[36]. Each inference rule consists of a set of statements (which have already been defined using the usual notation, $\vdash$, for sequent) called the *antecedents* and a statement called the *consequent.* In writing an inference rule, we separate antecedents and consequents by a horizontal line; axioms have no antecedents and no horizontal line is drawn.

We use the symbol A to represent the current type environment; A maps identifiers to types. We use the notation

$$A \vdash e : \tau$$

to indicate that expression e has type τ given the type bindings defined in A. Our initial type environment maps every identifier to, $\bot_t$, the least type in our type domain.

We reiterate our naming conventions: τ (and its subscripted variants) are used to range over types, δ (and its subscripted variants) range over type-schemes, and M and O (and their subscripted variants) range over the type and occurrence environments (resp.)

6.1.4.1 Semantics

If (M, O) is a map-type, then the expression $M[x \mapsto \tau]$ where τ is some type defines a new type environment as follows:

$$M[x \mapsto \tau](y) = \begin{cases} M(y) & \text{if } y \neq x \\ \tau & \text{otherwise} \end{cases}$$

A similar construction can be applied over O.

The constant function $M_\bot$ defines a type environment that maps every identifier to $\bot_t$; $M_\top$ is the constant function that maps every identifier to $\top_t$.

The constant function $O_{\mathbf{n}}$ maps every identifier to occurrence element $\mathbf{n}$; $O_\top$ maps every identifier to occurrence element $\top$.

In general, we shall use the notation:

$$M_s[n_1 \mapsto \tau_1, n_2 \mapsto \tau_2, \ldots, n_k \mapsto \tau_k]$$

to indicate the type environment that maps n_i to τ_i, $1 \leq i \leq k$ and maps all other elements to type s. A similar notation applies to occurrence environments as well.

6.1.4.2 The Type Rules

Type Axioms

There is one type axiom: $A \vdash x : A(x)$. This axiom states that the type of an identifier is the element to which it is mapped by the current type environment.

Instantiation and Specialization

There are two rules for instantiation and specialization of type variables.

(Type Variable Instantiation)

$$\frac{A \vdash e : \delta}{A \vdash e : \delta'} \qquad \delta > \delta'$$

(Generalization)

$$\frac{A \vdash e : \tau \qquad \alpha \text{ not free in } A}{A \vdash e : \forall\alpha\ \tau}$$

The first rule corresponds to type variable instantiation: given a type containing a type variable, we can always refine the type by substitution of free type variables for types in the type expression.

The second rule is a generalization rule that forms the basis for our polymorphic type system: given a type τ, we can make it a generic type by quantifying it over a new type variable. Thus, to make a type expression τ containing a free type variable α generic, we can replace τ with a type-scheme in which α is a bound variable. These two rules are, in fact, identical to the instantiation and generalization rules found in ML[24].

Type Rules for Environment-Yielding Expressions

There are two environment-yielding expressions in our restricted language: **map** and **select**.

(Map Introduction)

$$\frac{A[n_i \mapsto \tau_i] \vdash e_i : \tau_i, 1 \leq i \leq k}{A \vdash (\textbf{map}\ n_1 : e_1\ \ldots\ n_k : e_k) : (M_\perp[n_i \mapsto \tau_i], O_{\mathbf{n}}[n_i \mapsto \mathbf{a}])}$$

The map introduction rule allows mutually-recursive references within a given map expression; the type environment in which the subexpressions of a map are typed contains the type bindings of all names defined within the map. Thus, free references to names bound within the map will acquire the type of the expression to which that name is bound in the map. For example, the type of the map

```
(map
   x : y
   y : 3)
```

would be a pair, (M,O); M maps **x** and **y** to type **Integer** and all other identifiers to some type variable; O maps **x** and **y** to **a** and all other identifiers to **n**.

(Projection)

$$\frac{A \vdash e_1 : (M,O) \qquad M(x) = \tau_1, O(x) = \mathbf{a}}{A \vdash \texttt{(select } e_1\ x\texttt{)} : (M_\perp[x \mapsto \tau_1], O_{\mathbf{n}}[x \mapsto \mathbf{a}])}$$

The type of a projection expression is a map-type whose type environment maps the name being projected with its type in the original environment and whose occurrence environment maps the projected name to occurrence element **a**. Note that the antecedent requires that the projected name be present in the original environment. This constraint is consistent with our operational semantics: a projected name must be found in the environment onto which it is projected.

Scope-Expressions

We introduce a new operator, $\Longrightarrow$, in the type rule for a scope-expression that builds a new type environment from two existing ones. We give its definition here and explain the intuition behind the definition in the discussion following.

Definition 6.4 *Let M and A be type environments and let O be an occurrence environment. Then $O \Longrightarrow (M, A)$ is a type environment defined such that:*

$$(O \Longrightarrow (M,A))(x) = \begin{cases} M(x) & \textit{if}\ O(x) = \mathbf{a} \\ A(x) & \textit{if}\ O(x) = \mathbf{n} \\ M(x) \sqcup_t A(x) & \textit{if}\ O(x) = \top_o \\ \perp_t & \textit{if}\ O(x) = \perp_o \end{cases}$$

Given $\Longrightarrow$ we can express the inference rule for scope-expressions as follows:

(Scope-Expressions)

$$\frac{A \vdash e_1 : (M,O) \qquad O \Longrightarrow (M,A) \vdash e_2 : \tau_1}{A \vdash \texttt{(with } e_1\ e_2\texttt{)} : \tau_1}$$

The type of the scope-expression body is determined by the type-environment constructed by performing a composition operation on the lexical type environment A and the type environment associated with the type of the environment-specifier M. The structure of this new type environment is determined by the occurrence environment of the scope-expression: the type of a name that is *always* defined by e_1 is simply the type to which it is mapped in M; the type of a name that is *never* defined by e_1 is simply the type to which it is mapped in A; a name which has an inconsistent occurrence in M acquires a type that is the join of its type in M and A.

This rule captures our intuition about scope-expressions in the following sense: if a name is known to be defined by the environment-specifier, we can ignore its type in the lexical environment since it will never be used in determining the type of the scope-expression body. Similarly, if a name is known to be never defined by the environment-specifier, we are assured that its type in the body will be its lexical environment. The third condition applies if a name n is inconsistently defined by environment-specifier; in this case, we choose to make n's type in the scope-expression a supertype of its type in the lexical environment and its type in the environment-specifier (when it is present). This will make n's type in the scope-expression consistent with its type in the lexical environment and its type in the environment-specifier.

Type Rules for Abstraction and Application

(Abstraction)

$$\frac{A[x \mapsto \tau_1] \vdash e_1 : \tau_2}{A \vdash (\textbf{lambda}\ (x)\ e_1) : \tau_1 \rightarrow \tau_2}$$

The inference rule for lambda abstraction with slight modification can be used for defining the type of a kappa abstraction expression as well.

In words, this inference rule is to be read, "If, assuming a type environment A extended with a type binding τ_1 for x, we are able to derive the fact that expression e_1 has type τ_2, then the type of a lambda abstraction whose body is e_1 and whose formal is x is a function type from τ_1 to τ_2."

Notice that τ_1 is a free type in the inference rule; a type-checking procedure can assign any type to τ_1 provided that it satisfies the constraints on the use of **x** in the body of the abstraction. There may be many types that can be assigned to **x** that satisfy the necessary constraints; the type actually assigned is determined by the type-checking algorithm employed.

(Application)

$$\frac{\begin{array}{c} A \vdash e_1 : \tau_1 \rightarrow \tau_2 \\ A \vdash e_2 : \tau_3 \\ \tau_3 \sqsubseteq_t \tau_1 \end{array}}{A \vdash (e_1\ e_2) : \tau_2}$$

The application rule uses a subtype relation test that determines whether the type yielded by the actual is a subtype of the type of the formal. To see why such a test is needed, consider the following example:

```
f : (lambda (M)
        (select M x))
g : (f (map x:1 y:2))
h : (f (map))
```

Note that based on the inference rule for abstraction and projection, one possible type for **f** could be:

$$(M_{\top}[\mathbf{x} \mapsto \alpha], O_{\top}[\mathbf{x} \mapsto \mathbf{a}]) \rightarrow (M_{\perp}[\mathbf{x} \mapsto \alpha], O_{\mathbf{n}}[\mathbf{x} \mapsto \mathbf{a}])$$

Such a type assignment for **f** would allow it to be applied to any map-object that defines at least **x**. In other words, the minimal (*i.e.*, least constrained) type assignment for **M** that would still satisfy the inference rule for name projection would be one in which **M**'s occurrence environment maps **x** to **a**. The application of **f** to the map object defining names **a** and **b** is well-typed because the type of the actual is a subtype (*i.e.*, defines more names) of **M**'s type. On the other hand, the application of **f** to the empty map is not well-typed (as should be expected) since the type of the actual (here, $(M_{\perp}, O_{\mathbf{n}})$) is *not* a subtype of the formal.

Conditional and Selector Operations

(Conditional)

$$\frac{\begin{array}{c} A \vdash\ e_1 : \textbf{Boolean} \\ A \vdash\ e_2 : \tau_1 \\ A \vdash\ e_3 : \tau_2 \\ \tau_3 = \tau_1 \sqcup_t\ \tau_2 \end{array}}{A \vdash\ (\texttt{if}\ e_1\ e_2\ e_3) : \tau_3}$$

The arms of the conditional must be related together by the $\sqcup_t$ operation. To see why, consider the following example:

```
m : (if b
        (map x : 1)
        (map x : 1
             y : 2))
(with m y)
```

The type of the conditional is the join of its two arms; the type of the true arm is a map-type, (M_t, O_t) – O_t maps **x** to **a** and all other identifiers to **n**; M_t maps **x** to **Integer** and all other identifiers to $\bot_t$. The type of the false arm is also a map-type, (M_f, O_f) – O_f maps **x** and **y** to **a** and all other identifiers to **n**; M_f maps **x** and **y** to **Integer** and all other identifiers to $\bot_t$. The join of these two types is a map-type, (M_j, O_j) where O_j maps **x** to **a**, **y** to $\top_o$ and all other names to **n** and M_j maps **x** to **Integer** and **y** to **Integer**. If **m** is subsequently used in a scope-expression in which **y** is free in the body, **y**'s type in the evaluation environment of the scope-expression must be compatible with **Integer** by virtue of the $\Longrightarrow$ operator; an incompatible type in the lexical environment will result in **y**'s type in the body being $\top_t$.

The Symmetric Lisp type rule for conditionals differs from ML's because of its join rule. Note, however, that the type system does not support dependent types[49]: the type of a conditional whose arms are of unrelated type is always $\top_t$. Practical experience with the language indicates that one rarely needs to define objects of completely different type in different arms of the conditional. In particular, it is hardly ever the case that one needs to define a conditional in which the two arms define map objects which are unrelated to one another.

(Selection)

$$\frac{A \vdash e_1 : (M, O) \qquad A \vdash e_2 : \textbf{Integer}}{A \vdash e_1[e_2] : \bigsqcup_t M}$$

The type of a region selection operation is computed by taking the join of all types mapped by the type environment associated with the argument map's map-type. Readers will note the strong similarity between this type rule and the type rule for conditionals. In the case of the conditional, it is necessary to compute the join of the two arms; in the case of the selector, it is necessary to compute the join of all types in M's range.

6.1.5 Examples

The type inference rules given in the previous section constitute a set of *constraints* on a program which, if satisfied, implies that the program is well-typed. In principle, a type-checking algorithm based on these inference rules will set up a constraint system and then solve it with respect to the free variables found therein. There will often be several possible solutions to a set of constraints; the one actually constructed is a property of the type-checking procedure used.

Example 1

Consider the program fragment:

```
y : nil
f : (lambda (x) (with x y))
(f (map
     y : 3))
```

The type constraints for this program fragment are:

	Initial Assumption	
[1]	A	$M_\bot$
	Expression	*Type*
[2]	`nil`	**Boolean**

[3]	**y**	**Boolean**
[4]	**x**	(M, O)
[5]	**y**	α
[6]	`(lambda (x) ...)`	$(M, O) \rightarrow \alpha$
[7]	**f**	$(M, O) \rightarrow \alpha$
[8]	3	**Integer**
[9]	**y**	**Integer**
[10]	`map y : 3`	$(M_\perp[\mathbf{y} \mapsto \mathbf{Integer}], O_\mathbf{n}[\mathbf{y} \mapsto \mathbf{a}])$
[11]	`(f (map ...))`	α

Additional Constraints

[12] $(O \Longrightarrow (M, A[\mathbf{y} \mapsto \mathbf{Boolean}]))(\mathbf{y}) = \alpha$
[13] $(M_\perp[\mathbf{y} \mapsto \mathbf{Integer}], O_\mathbf{n}[\mathbf{y} \mapsto \mathbf{a}]) \sqsubseteq_t (M, O)$

Line [1] defines the initial type environment. Lines [2] and [3] define the type for the outermost binding of **y**. Line [4] constrains **x** to be a map-type based on its use in the scope-expression. Line [6] defines the type of the lambda. Line [10] constrains the type of `y : 3` to be a map-type where **y**'s type is **Integer** and **y**'s occurrence element is **a**. The type of the application is given in line [12].

Besides these constraints, there are two additional subtyping constraints. Line [12] constrains M (the type-environment for **x**) and A (the lexical environment) under the $\Longrightarrow$ constructor; it states that the M and A must have a type such that **y** has type α under $\Longrightarrow$. Line [13] defines the sutype constraint between the type of the actual parameter in the application,

$$(M_\perp[\mathbf{y} \mapsto \mathbf{Integer}], O_\mathbf{n}[\mathbf{y} \mapsto \mathbf{a}])$$

and **x**'s type, (M, O); this contraint specifies that the actual's type must be a subtype, *i.e.*, must be more constrained than the type of the formal.

We can consider a set of constraints as a being nothing more than a set of equations with some free type variables; the role of the typechecker is to solve this set of equations with respect to these type variables. In the above example, M, O, and α are the free type variables. One meaningful assignment of these variables would be to instantiate M to $M_\perp[\mathbf{y} \mapsto \mathbf{Integer}]$ and O to O to $O_\mathbf{n}[\mathbf{y} \mapsto \mathbf{a}]$. Given this assignment for M and O, α must get instantiated to be **Integer**. Such an assignment would fail if there weio a subsequent application of **f** in which y was bound to a string; the actual would not be a subtype of (M, O) in this case.

Note also that a type assignment in which O is instantiated to be $O_{\mathbf{n}}$ would not satisfy the subtype constraint imposed by line [13]. Such an assignment would be caught by the type-checker since it would violate the subtyping constraints over map-types.

Example 2

We consider another related example:

```
e : ((lambda (x)
       (with x
         (map w : y)))
     (map y : 2
          z : 4))
```

One could define a system of type constraints for this expression as follows:

	Initial Assumption	
[1]	A	$M_\perp$
	Expression	*Type*
[2]	2	**Integer**
[3]	4	**Integer**
[4]	`(map y:2 z:4)`	$(M_\perp[\mathbf{y} \mapsto \textbf{Integer}, \mathbf{z} \mapsto \textbf{Integer}]$ $O_{\mathbf{n}}[\mathbf{y} \mapsto \mathbf{a}, \mathbf{z} \mapsto \mathbf{a}])$
[5]	`x`	(M,O)
[6]	`y`	α
[7]	`(map w : y)`	$(M_\perp[\mathbf{w} \mapsto \alpha], O_{\mathbf{n}}[\mathbf{w} \mapsto \mathbf{a}])$
[8]	`(lambda (x) ...)`	$(M,O) \rightarrow (M_\perp[\mathbf{w} \mapsto \alpha], O_{\mathbf{n}}[\mathbf{w} \mapsto \mathbf{a}])$
[9]	`e`	$(M_\perp[\mathbf{w} \mapsto \alpha], O_{\mathbf{n}}[\mathbf{w} \mapsto \mathbf{a}])$

Additional Constraints

[10] $(O \Longrightarrow (M, A[\mathbf{x} \mapsto (M,O)]))\,(\mathbf{y}) = \alpha$

[11] $(M,O) \sqsupseteq_t (M_\perp[\mathbf{y} \mapsto \textbf{Integer}, \mathbf{z} \mapsto \textbf{Integer}], O_{\mathbf{n}}[\mathbf{y} \mapsto \mathbf{a}, \mathbf{z} \mapsto \mathbf{a}])$

Line [1] specifies the constraint on the initial environment. Lines [2] and [3] express the constraints for the predefined integer constants 2 and 4. The type axiom and map introduction rule impose the type constraint on the argument map expressed in line [4]; it states that the type of the map

```
(map y : 2
     z : 4)
```

is a map-type, $(M_\perp[\mathbf{y} \mapsto \textbf{Integer}, \mathbf{z} \mapsto \textbf{Integer}], O_{\mathbf{n}}[\mathbf{y} \mapsto \mathbf{a}, \mathbf{z} \mapsto \mathbf{a}])$.

Lines [5] – [7] specify the type constraints imposed by the scope-expression. Line [5] constrains **x** to be a map-type and line [6] sets the type of the free variable **y** in the scope-expression body to be a type variable α. Line [7] constrains the type of the scope-expression body based on the type axiom and the map introduction rule.

Line [8] specifies the type of the lambda abstraction and line [9] gives the type of the application; this constraint is derived using the application rule based on the type inferred for the abstraction from line [8].

Lines [10] and [11] are the subtyping constraints given in the inference rule for scope-expressions and the application rule. Line [10] places a constraint on both M and $A[\mathbf{x} \mapsto (M, O)]$, the current lexical environment, by associating the type of **y** to be α in the type environment yielded by the $\Longrightarrow$ constructor. Line [11] specifies the subtype constraint between the actual and the formal in the application.

Typechecking **e** consists in (1) showing that the system of constraints is consistent (*e.g.*, it should not be possible to show that an **Integer** type = a map-type), and (2) solving the constraints with respect to α, M and O which are the three free variables defined in the equations.

There are several possible solutions to this constraint system. One trivial solution would be to simply set M to $M_\top$ and O to $O_\top$. α would then become the join of $\top$ and whatever the type of **y** is as specified by A; by definition of the $\sqcup_t$ operator, this type will always be $\top_t$. The type of **e** is then a map-type that associates **w** with $\top_t$ and all other elements with $\perp_t$. This is certainly a "correct" solution insofar as it satisfies all the necessary type constraints, but it is not a very useful one since it unnecessarily overconstrains **y**. Given this type assignment, the map object bound to **e** can never be treated as a subtype of one that has a non-$\top_t$ type assignment for **y**.

A more useful approach to assigning types would be one that imposes the minimum constraints on the free type variables while still satisfying all the necessary conditions. For example, consider a solution that assigns $M_\perp[\mathbf{y} \mapsto \textbf{Integer}, \mathbf{z} \mapsto \textbf{Integer}]$ to M and $O_{\mathbf{n}}[\mathbf{y} \mapsto \mathbf{a}, \mathbf{z} \mapsto \mathbf{a}]$ to O. The type of α now becomes constrained to be **Integer** as per the definition of the $\Rightarrow$ function. This assignment also constitutes a valid solution since no constraints are violated, but it imposes a minimum constraint on M, O, and α – it is the most general (or principal) type for the above expression and arguably the most useful. A type-checking algorithm that computes the principal type computes the least solution to a series of constraint equations.

Example 3

We now consider an example involving **select** and higher-order functions:

```
f : (lambda (x)
      (if x
          (lambda (g) (select g p))
          (lambda (h) (select h q))))
```

f defines a higher-order function that given a boolean returns either a function that selects name p from its argument map or a function that selects name q from its argument.

The system of type constraints for this function can be given as follows:

	Initial Assumption	
[1]	A	$M_\perp$
	Expression	*Type*
[2]	`x`	**Boolean**
[3]	`(select g p)`	$(M_\perp[\mathrm{p} \mapsto \alpha_1], O_{\mathbf{n}}[\mathrm{p} \mapsto \mathbf{a}])$
[4]	`g`	(M_1, O_1) where $M_1(\mathrm{p}) = \alpha_1$ and $O_1(\mathrm{p}) = \mathbf{a}$
[5]	`(lambda (g) (select g p))`	$(M_1, O_1) \rightarrow (M_\perp[\mathrm{p} \mapsto \alpha_1], O_{\mathbf{n}}[\mathrm{p} \mapsto \mathbf{a}])$
[6]	`(select h q)`	$(M_\perp[\mathrm{q} \mapsto \alpha_2], O_{\mathbf{n}}[\mathrm{q} \mapsto \mathbf{a}])$
[7]	`h`	(M_2, O_2) where $M_2(\mathrm{q}) = \alpha_2$ and $O_2(\mathrm{q}) = \mathbf{a}$
[8]	`(lambda (h) (select h q))`	$(M_2, O_2) \rightarrow (M_\perp[\mathrm{q} \mapsto \alpha_2], O_{\mathbf{n}}[\mathrm{q} \mapsto \mathbf{a}])$
[9a]		$(^*) = (M_\perp[\mathrm{p} \mapsto \alpha_1], O_{\mathbf{n}}[\mathrm{p} \mapsto \mathbf{a}]) \sqcup_t (M_\perp[\mathrm{q} \mapsto \alpha_2], O_{\mathbf{n}}[\mathrm{q} \mapsto \mathbf{a}])$
[9b]	`(if x ...)`	$(M_1, O_1) \sqcap_t (M_2, O_2) \rightarrow (*)$
[10]	`f`	**Boolean** $\rightarrow ((M_1, O_1) \sqcap_t (M_2, O_2) \rightarrow (*))$

Line [1] assumes the intial type environment is $M_\perp$. Line [2] is derived from the inference rule for conditionals. Lines [3] and [4] define the constraints imposed by the projection rule. Line [3] states that p has type α_1 (where α_1 is a free type variable); Lines [4] constrains g to be a map-type such that its type environment map p to α_1 and its occurrence environment map p to a. Line [5] is the type constraint introduced by the abstraction rule. Lines [6] – [8] are rules similarly defined for the false arm of the conditional. Lines [9a] and [9b] impose the type constraints specified by the conditional: the type of the conditional is the join of the types of its arms. Finally, line [10] defines the constraints imposed by the conditional inference rule and the abstraction rule. (Were we absolutely precise, we would have also introduced a separate type variable for x; the constraint system would instantiate this variable to type **Boolean**.)

There are six free type variables in this constraint system. Rules [4] and [7] constrain the type variables introduced for the map-types of the two lambda expressions. Here again, there are many possible assignments for the free variables that would satisfy the constraints. The most general type assignment would associate the following types to the map-type variables:

$$(M_1, O_1) = (M_\top[\mathtt{p} \mapsto \alpha_1], O_\top[\mathtt{p} \mapsto \mathbf{a}])$$

and

$$(M_2, O_2) = (M_\top[\mathtt{q} \mapsto \alpha_2], O_\top[\mathtt{q} \mapsto \mathbf{a}])$$

This type assignment imposes a minimum constraint on the occurrence environment of g and h since it enforces no restriction on the occurrence of any identifier outside of the one actually being selected.

Given this assignment, the join of the two functions can be computed. The join of two functions is determined by taking the meet of their domains and the join of their ranges. Given the above assignment for g and h, we can compute their meet to be:

$$(M_\top[\mathtt{p} \mapsto \alpha_1, \mathtt{q} \mapsto \alpha_2], O_\top[\mathtt{p} \mapsto \mathbf{a}, \mathtt{q} \mapsto \mathbf{a}]) \qquad (*)$$

Notice that the function returned by f can be applied to any map object that defines p and q and that the type of p and q in the argument map is unconstrained by the type definition.

A type assignment for the conditional can now be derived:

$$* \rightarrow (M_\perp[\mathtt{p} \mapsto \alpha_1, \mathtt{q} \mapsto \alpha_2], O_{\mathbf{n}}[\mathtt{p} \mapsto \top_o, \mathtt{q} \mapsto \top_o])$$

We can instantiate α_1 and α_2 to be type-schemes by making them bound variables of a universal quantifier defined at the outermost level of the above type. This assignment makes makes p and q generic objects: their types are totally unconstrained.

What can we infer about this assignment? Clearly, it allows the function returned by f to be applied to any function that defines at least p and q. Moreover, the type of the result yielded by this function states that both p and q are defined inconsistently – it cannot be guaranteed that either identifier is found in the result.

6.1.6 Incorporating Assignment and Empty Regions

Empty regions and assignment complicate the type rules. The fundamental problem has to do with the interaction of polymorphism and mutable objects. To see why this is an issue, consider the following expression that can be constructed using the full language:

```
f : (lambda ()
      x : (map *)
      g : (lambda (a b)
            (if a
                (fill x[1] b)
                x[1])))
h : (f)
```

What should h's type be? The type of the object returned by f is a function that given some boolean a and some polymorphic object b, side-effects the first element of an own variable map x defined in f with b if a is true and returns the first element of x if a is false. Notice that the type of x[1] should be the same type as b since the fill operation fills x[1] to be b if a is true.

One possible type we might expect a type-checker for the full language to associate with h would be of the form:

$$\textbf{Boolean} \times \alpha \rightarrow \alpha$$

Intuitively, this type assignment says that given a boolean and an object of some type α, h will return some object of type α. (This assumes that the fill expression constrains the type of the region being filled to be the same as the type of the object which is going to occupy it.)

Now consider the effect of evaluating the following two expressions in sequence:

```
(h t t)
```

and

```
(+ (h nil 5) 6)
```

In a fully polymorphic type-system such as the one given in this chapter, both applications would be considered well-typed; the first instantiates α to be a boolean, the second instantiates α to be an integer. Note, however, that the first application performs a side-effect on x by filling x's first region with a boolean value. When the second application is evaluated, the value of x[1] will be the boolean, t, not an integer as is expected. When the addition operation is performed, a runtime type error would result.

The basic problem has to do with the *genericity* of α; because of the presence of side-effects in g, we need to make α non-generic across applications. Thus, the type of f after h is first applied should become constrained to be

Boolean $\times$ **Boolean** $\rightarrow$ **Boolean**

Subsequent applications of h should conform to this type. Given the ability to distinguish between generic and non-generic type variables, the second application of h above would raise a type-error as is to be expected. In the type system presented thus far, however, functions are instantiated generically; the effect of updating a storage location is not reflected in the expression's type.

It is possible to handle empty regions and general assignment in a polymorphic type system, but doing so involves augmenting the type system so that it keeps track of objects which are mutable and those which are not. For a more comprehensive account of how one might handle mutable structures in a polymorphic type-system, the reader should see [63]. An alternative approach using *effect inferencing* is presented in [37]; effect inferencing can be used to determine the observable side-effects that an expression may have when evaluated.

6.1.7 Soundness

The inference system given in this chapter constitutes a syntactic inference system – the inference rules are structured based on the *form* of the expression; it makes no assertion on the relationship between an expression's *meaning* or *denotation* and its type. In order to formally show that the inference rules capture our notion of what a well-typed program should be, we need to define a denotational semantics for the expression language that relates expressions to elements in an abstract mathematical space. We can then construct a precise definition of what we mean by a well-typed program by treating a type as defining a subset in this abstract space. We do not define a formal denotational semantics in this thesis; we leave it to the reader's intuition to convince himself that the inference rules given above do capture the notion of what we mean by a well-typed expression.

We can, however, sketch the basic structure of the formal semantics as follows; the actual construction of the semantic clauses is omitted. The basic semantic function $\mathcal{E}$ has the following domain equation:

$$\mathcal{E} : Exp \to Env \to V$$

where Exp defines the category of expressions and $Env = Id \to V$ defines the environment function that maps identifiers to their abstract values; V is the value domain. We assume that there are two special values in V called $\perp_{vt}$ and $\perp_{ve}$ used to model runtime *type* errors (*e.g.*, using an integer as a function, etc.) and runtime exceptions (*e.g.*, asking for the 10^{th} element in a 9 element map, etc.), resp.

Intuitively, a well-typed program will never return $\perp_{vt}$ at runtime. For example, a projection expression that attempts to select a name x from some map that does not define x will acquire value $\perp_{vt}$; this error should be captured by a type-checker that implements the type rules. Similiar reasoning applies to all occurrences of $\perp_{vt}$ in the formal semantics: $\perp_{vt}$ is to be thought of a runtime type error that can be captured at compile-time by a type-checking system.

A type is thought of as a set of values; every element in the value domain *except* $\perp_{vt}$ and $\perp_{ve}$ is an element of some subset of V that does not contain $\perp_{vt}$ or $\perp_{ve}$. We define $\mathcal{D}$ to be the semantic function that maps type expressions to the set of elements in the value domain which the type denotes:

$$\mathcal{D} : Type \to \mathcal{P}(V)$$

If it is possible to deduce that e has type τ, then the value denoted by e belongs to the domain denoted by τ. Occurrences in this framework are projections on types; they are used to select a subset of the set of elements denoted by a type.

Following Cardelli[16], we say that a type environment A *agrees* with a semantic environment S if for any x, $S[\![x]\!] \in \mathcal{D}[\![\tau]\!] \Longleftarrow A \vdash x : \tau$. We relate the semantics of SL_O with the syntactic type rules in the following conjecture:

Conjecture 6.1 *If S agrees with A, and $A \vdash e : \tau$, then $\mathcal{E}[\![e]\!]S \in \mathcal{D}[\![\tau]\!]$*

The proof of this theorem and the construction of the semantic domains are left for future research.

Chapter 7

Compiling Into a Dataflow Language

The Symmetric Lisp code-generator has the responsibility for translating Symmetric Lisp expressions into a suitable base language. The parallel semantics of maps imposes two non-trivial requirements on the base language. First, because a map is a parallel process creator (evaluating a map with n elements creates n parallel threads of computation) it is necessary that the base language provide efficient support for fine-grained parallelism. Secondly, because a map is a non-strict data structure (a map object is available for inspection even if not all of its elements have finished computing) it is necessary that the base language provide efficient synchronization mechanisms to support non-strictness.

Because of the need to have base language support for fine-grained parallelism and non-strictness, we choose to compile Symmetric Lisp programs into the language of *dynamic dataflow graphs*[6]. Dataflow graphs have several important strengths that are especially useful in implementing Symmetric Lisp.

1. Dataflow graphs impose minimum constraints on parallelism: the execution of a dataflow instruction is constrained only by data dependencies; the execution semantics of a dataflow language imposes a *partial* (rather than a total) order on instruction execution. As a consequence of this property, every instruction in a dataflow program is potentially executable, capable of executing concurrently with any other instruction whose data dependencies have been satisfied.

2. It is straightforward to provide support for non-strict data structures in the context of a dataflow execution model. An instruction which accesses an element of a non-strict data structure whose value is still being computed[1] blocks until the value becomes available.

3. The compilation of high-level languages into dataflow graphs has been well-studied [26, 64]. The translation of high-level constructs such as let-blocks, functions, application etc.

[1]Such elements are typically implemented using I-structures[7] or early-completion queues[25, 26]

follow very straightforward translation rules that are amenable to significant optimizations.

There are two approaches that can be taken to describe the translation of Symmetric Lisp expressions into a dataflow graph base language. The first approach would entail exhibiting a direct translation from Symmetric Lisp into actual dataflow schemata. Presenting the translation rules this way has the advantage of explicitly relating Symmetric Lisp expressions to their dataflow graph counterparts, but the discussion would necessarily involve introducing a significant amount of machinery to describe the semantics of the dataflow graph language. An alternative approach and the one we choose in this chapter is to exhibit a source-to-source transformation from Symmetric Lisp expressions into another high-level language for which the translation into dataflow graphs is already well-understood. This approach has a significant advantage over the direct translation scheme since we avoid having to deal with the relatively low-level language of dataflow graphs; the exposition (presumably) will be correspondingly easier to understand.

The language we choose as our target language is Id[53], a high-level functional language augmented with a parallel data-structuring mechanism called I-structures. Like Symmetric Lisp, the semantics of Id specifies that all data structures and applications be non-strict. The language supports abstract data types, powerful composition constructs for lists and arrays, general loops, and curried higher-order functions. For a detailed exposition on Id and its implementation, the reader is referred to [8, 53]; the translation of Id into an abstract dataflow graph representation is given in [64]. In describing the translation, we will sometimes require operations which have no straightforward implementation in Id. We implement these operations either as abstractions or as new primitives.

The ability to transform Symmetric Lisp into Id implies that Symmetric Lisp and Id are equal in expressive power in a weak Turing-equivalence sense: while the semantics of the Id representation of a Symmetric Lisp program may be observationally faithful to the semantics of the Symmetric Lisp version, it should be kept in mind that the programming style, methodology and particular paradigms encouraged by Symmetric Lisp are far different from that of Id.

For the sake of clarity, our translation performs no optimizations on either the source or target code; as the reader will note, there is much room for optimization in the Id code generated by the transformation operation. We also do not assume that presence of an environment inferencing algorithm; consequently, evaluating free variables in scope-expressions will necessarily involve traversing an environment chain. We assume that there is no type-checking on the generated Id program: thus, it may well be the case that the translated code shown in this chapter is not type-correct according to the type rules governing Id expressions. Finally, our translation does not consider Symmetric Lisp programs containing the **read** primitive. The set of identifiers defined in programs containing **read** cannot be statically determined; the translation rules we present here depend upon the ability to statically compute the set of all identifiers found in a Symmetric Lisp program. We come back to this issue in Section 7.5.

7.1 Symbol Tables

A Symmetric Lisp map has no direct counterpart in Id. To help implement maps in Id we introduce a new Id data abstraction called a *symbol table.* An instance of a symbol table is a structure that associates strings to either integers or the special symbol, **undef**. The set of strings in a symbol table S is referred to as S's *domain* and the set of integers to which these strings are associated is referred to as S's *range.* The symbol table will serve as a repository for the names defined within a map. It will be the primary structure used in the implementation of scope-expressions. There are a number of possible representations for a symbol-table of this kind; the most efficient representation in terms of access time would be a hash-table structure. We leave the construction of the symbol table abstraction in Id as an exercise for the reader.

There are four main operations over symbol-tables:

1. **make_symbol_table**: Given a list of pairs, <*string,integer*>, **make_symbol_table** returns a symbol table containing a binding for each of the pairs input.

2. **names_in_table**: Given a symbol table S, **names_in_table** returns a list containing the strings comprising S's domain.

3. **get_val**: Given a symbol table S and a string s, **get_val** returns the integer bound to s in S if s is in S's domain or **undef** otherwise.

4. **get_symbol**: Given a symbol table S and an integer i, **get_symbol** returns a string s such that (**get_val** S,"s") = i. If no such string exists in S's domain, **get_symbol** returns **undef**.

7.2 The Top-Level

A Symmetric Lisp program P_{SL} is translated into a corresponding Id program P_{Id} that has the following top-level structure:

```
{ id_1 = error;
  id_2 = error;
     ...
  id_k = error
in
  < translated version of P_SL>}
```

Every identifier in P_{SL} is bound at the top-level to the special symbol, **error**. Thus, any reference to an identifier in the translated version of P_{SL} that is not defined in the evaluation environment of that identifier will eventually get value **error**. The value of an Id program in which an identifier reference yields **error** is assumed to be **error** also.

7.3 Map Expressions

A map expression serves two roles in Symmetric Lisp. As a program structure is acts a local naming environment; as a data structure it plays the dual role of both a heterogeneous record as well as a vector. Id provides direct support for local naming environments (as program structures) via the *block* construct. The view of maps as data structures can be expressed using symbol tables.

For example, the following map expression

```
(map n1 : e1
     n2 : e2
      :
     nk : ek)
```

(where each of the `ei` are regions) can be translated into the following Id program fragment:

```
{ a = array(1,k);
  a[1] = < translated version of e1>;
  a[2] = < translated version of e2>;
    ...
  a[k] = < translated version of ek>
  n1 = (a,1);
  n2 = (a,2);
    ...
    nk = (a,k)
  in (a,(make_symbol_table (("n1",1) ("n2",2) ... ("nk",k))))}
```

a is a fresh name guaranteed not to occur free in any of the ei. The above Id fragment evaluates each of the k expressions defined in the Symmetric Lisp map in a local environment; because blocks in Id are mutually-recursive, free references to any of the ni made by any of the ei are resolved in a manner consistent with the name-lookup rule for Symmetric Lisp identifiers. We describe the exact translation of identifiers below.

Note that identifier ni in the block is bound to a tuple whose first component is the array representation of the map being constructed and whose second component is the index into the array that contains ni's value. This tuple effectively represents ni's *l-value*; it defines the *address* (rather than the *value*) of the region denoted by ni. We show how to derive the *r-value* of a named-region below. The reason for binding identifiers to l-values rather than r-values is due to the fact that regions can be side-effected; by recording the l-value of an expression, it becomes straightforward to side-effect a region when necessary. Expressions which require the r-value of an identifier must explicitly destructure the tuple representation of the l-value. The l-value contains the information to retrieve the identifier's r-value.

The result of the block is a two-tuple consisting of a k-element array a and a symbol-table. The i^{th} element in the array contains the value of the i^{th} element in the Symmetric Lisp map; the symbol-table associates each name (represented as a string) defined in the map with an integer. This integer represents the index in a that contains the binding-value of the Symmetric Lisp identifier in the map.

In the above fragment, every region expression is named and appears on the right-hand side of the binding statement whose left-hand side is the name of the region; a map expression whose i^{th} region (say e) is unnamed would be translated by binding a[i] directly to e; i would not be in the range defined by the symbol-table associated with a.

Because arrays are non-strict, the result array can be returned as the result of the block even as expressions within it continue to evaluate. The two elements of the tuple structure, the array a and the symbol-table embody the two data-structure related facets of Symmetric Lisp maps: the array allows map elements to be selected by position; the symbol table will be used to support name-based selection. Given this representation for maps, we now proceed to describe the translation of the other fundamental Symmetric Lisp operators.

7.4 Translating Identifiers

The translation rule for identifiers depends upon whether their l-value or r-value is needed. We will see how to use the l-value of an identifier when we examine assignment operations; for now, we restrict ourselves to determining the r-value of an identifier.

The map expression:

```
(map
   y : 1
  (map
     x : 2
     (+ x y)))
```

would be translated into the following Id program fragment:

```
{  a = array(1,2);
   a[1] = 1;
   y = (a,1);
   a[2] = { b = array(1,2);
            b[1] = 2;
            b[2] = { (arr_1,index_1) = x;
                     (arr_2,index_2) = y;
                    in
                      arr_1[index_1] + arr_2[index_2] };
            x = (b,1)
          in
            (b,make_symbol_table(("x",1))) }
 in
   (a,make_symbol_table(("y",1))) }
```

(We assume that the "+" operator is not redefined elsewhere in the lexical scope of this fragment.) The plus operation destructures the tuple that defines the l-value for **x** and **y** and uses the information provided therein to access their r-values.

7.5 Scope-Expressions

Symbol-tables are used in the implementation of scope-expressions. Consider a Symmetric Lisp expression of the form

```
(with e1 e2)
```

Let n1,n2,...,nk occur free in e2. (For a precise description of what we mean by "free", the reader is referred to page 59.)

We can exhibit a translation into an Id program fragment that would capture the behaviour of the above scope-expression as follows:

```
{  (a,s) = < translated version of e1>
   v1 = (get_val s "n1");   x1 = n1;
   v2 = (get_val s "n2");   x2 = n2;
          ...
   vk = (get_val s "nk");   xk = nk;
 in
   {  n1 = (if v1 =  undef then x1 else (a,v1));
      n2 = (if v2 =  undef then x2 else (a,v2));
                    ...
      nk = (if vk =  undef then xk else (a,vk))
     in < translated version of  e2>}}
```

We assume that a, s, vi and xi are fresh variables. The evaluation of the environment-specifier yields a tuple (as per the translation of maps given in the previous section); the second element of this tuple is a symbol-table containing the names defined by the map. Each xi is bound to the value of ni in the lexical environment of the scope-expression.

The body of the scope-expression is evaluated in an environment in which each free name is bound to either its value in the scope-expression's lexical environment or to its value in the environment yielded by the environment-specifier (*i.e.*, if ni is a free variable in the body, then it is either bound to its value in the map yielded by evaluation of the environment-specifier or its value in the scope-expression's lexical environment).

Suppose that a free name is defined by the environment-specifier but is not defined in the lexical environment of the scope-expression. Thus, for example, it may be the case that n1 is defined by the map yielded by e1 but is not defined in the lexical environment. The reference to n1 in the above skeleton would propagate up to the top-level. Recall that the top-level block binds each identifier referenced in the program to a special error value; as a result, x1 will get bound to the symbol **error**. If n1 is not defined by e1, any reference to it made in the body of the scope-expression will result in an error being raised. The correctness of this translation relies on the assumption that the set of free names in the body of the scope-expression can be determined. This assumption is valid in the subset we are considering here, but may not hold in the presence of operators such as **read** which may coerce arbitrary text into bona fide map objects. Thus, the program fragment:

```
(with (map x: 1) (+ x (mlast (read io-stream))))
```

where `(read io-stream)` returns `map y` will result in a Id (as opposed to Symmetric Lisp) runtime error if `y` is not defined in the lexical environment of the scope-expression. Expressions like `read` that can dynamically introduce new names into a program are best interpreted. Although Id can handle the lexical name-lookup rule found in Symmetric Lisp, there is no convenient underlying environment structure visible to the programmer that can be used to dynamically search for a variable; such a structure must be built on top of Id via appropriate abstractions.

If a free name is not bound in the environment-specifier, its value in the specifier's symbol-table would be **undef**; a name bound to **undef** takes its value in the lexical environment. (This implementation assumes, of course, that **undef** is a symbol guaranteed not to be used as the binding value of a name in the Symmetric Lisp map.)

Each of the `ni` in the inner block are bound to l-values; thus, side-effects on these identifiers in `e2` will be visible outside; in particular, if `e2` side-effects an identifier defined in `n1`, the side-effect will be visible to other expressions that access the array containing the values of `e1`'s regions.

For example, given a Symmetric Lisp program fragment of the form:

```
(map
   f : (lambda (foo bar) ... )
   (with (map x : 1
              y : 3)
      (f x y)))
```

the corresponding Id translation would be of the form:

```
{ a = array(1,2);
   a[1] = < translation of (lambda (foo bar) ... )>;
   a[2] = { (a,s) = {  a2 = array(1,2);
                       a2[1] = 1;
                       a2[2] = 3;
                       x = (a2,1);
                       y = (a2,2)
                      in (a2,make_symbol_table (("x",1) ("y",2))) };
               v1 = (get_val s "f");  x1 = f;
               v2 = (get_val s "x");  x2 = x;
               v3 = (get_val s "y");  x3 = y
             in {  f = (if v1 =  undef then x1 else (a,v1));
                   x = (if v2 =  undef then x2 else (a,v2));
                   y = (if v3 =  undef then x3 else (a,v3))
                  in {  (arr,index) = f
                        in (< translation of the application> }}}
   f = (a,1)
 in (a,make_symbol_table (("f",1))) }
```

In this example, the body of the scope-expression contains three free variables, `f`, `x` and `y`. The environment-specifier defines a map containing two regions; the first region contains a binding for `x` and the second region contains a binding for `y`. `f` is a function defined in the scope-expression's lexical environment.

The exact translation of lambda expressions is shown in Section 7.8 but suffice it to say for now that Symmetric Lisp lambda expressions are translated into Id functional objects.

There is ample room for improvement in the translation. Given an implementation of the environment inferencing algorithm described in the last chapter, none of the conditional tests for **undef** in the above fragment would be necessary; the inference procedure would determine that the environment-specifier always defines names `x` and `y` and never defines name `f`. The code generated would bind `f` to `x1` and `x` and `y` to `v2` and `v3` resp.

7.6 The Layer Operation

7.6.1 Comprehension Statements in Id

Before presenting the translation of the layer operator, we briefly explain some of the array and list defining constructs found in Id that we use extensively in the translation. A more detailed description may be found in [53].

A *list comprehension* statement (as used in the examples below) is written:

```
{: e || GEN1 & GEN2 & ... GENm }
```

where a generator is written

```
pat <- el FILTER1 ... FILTERn
```

Each `el` evaluates to a list and `pat` is matched to each element in the list, generating a list of environments that bind the pattern variables. In Id, each `el` may also be applied to a *filter* which is either written `when epw` or `unless epu`. Those environments in which `epw` evaluates false or an `epu` evaluates true are discarded.

The expression `e` is evaluated in each environment produced by the generator and the values yielded are collected into a list (in the same order) which is the result of the whole expression.

There may be more than one generator in a list comprehension. Each generator is evaluated left to right; the net result of the generator sequence is a sequence of environments containing bindings for the pattern variables of all the generators.

An *array comprehension* statement (as used in the examples below) is written:

```
{ array (lower-bound, upper-bound)
      | [e11] = e12 || GEN
      | ...
      | [eM1] = eM2 || GEN }
```

The array comprhension statement behaves similarly to the list comprhension form except that the `ej1` are integer expressions; the `ej2` produce the value to be stored in the index specified by `ej1`. The generators produce a sequence of environments in which the `ej1` and `ej2` are evaluated. All clauses are evaluated simultaneously and the top-to-bottom order of the clauses has no signficance. We note that the more general form of an array comprehension statement allows arbitrary n-dimensional arrays to be built.

7.6.2 Translating the Layer Operation

The translation of the layer operation is straightforward. The expression (`layer` M N) can be translated into Id as follows:

```
{ (a1,s1) = < translation of M >;
  (a2,s2) = < translation of N >;
  (1,j) = bounds a1;
  (1,k) = bounds a2;
  names_in_M = (names_in_table s1);
  names_in_N = (names_in_table s2);

  N_M_intersection = intersection names_in_M names_in_N;
  N_M_intersect_indices = {: (get_val s2 s) || s <- N_M_intersection};
  intersection_size = length N_M_intersection;
  count_indices l i = {  f x y = if x > i then 0 else 1;
                        in foldl_list (+) 0 (map_list f l) };

  N_M_dif = difference names_in_N names_in_M;
  N_M_dif_indices = {: x || x <- 1 to k unless (member x N_M_intersect_indices)};

  layered_array =  { array (1,j+k-intersection_size)
     | [t] = a1[t] || t <- 1 to j
     | [t] = a2[x]
               || t <- j+1 to j+k-intersection_size & x <- N_M_dif_indices }
  M_symbols = {: (s,(get_val s1 s)) || s <- names_in_M};
  N_symbols = {: (s,j + (get_val s2 s) -
                      count_indices N_M_intersect_indices
                                  (get_val s2 s)) || s <- N_M_dif }
```

```
 in
    (layered_array,(make_symbol_table (union M_symbols N_symbols)))}
```

We assume that binding-names introduced in the block are fresh names that do not conflict with names found in the Symmetric Lisp program.

To determine the structure of the map yielded by the layer operation requires determining the intersection of names defined by the maps being layered. Let the two maps be M and N. If a name x appears in both maps, we discard the binding associated with x in N in the layered map.

The **layered-array** is an array whose size is the sum of the sizes of the array representation of the two arguments minus the number of elements that are defined with the same name in the two maps. Thus, given a map M (whose array representation contains m elements) and a map N (whose array representation contains n elements) and where both M and N have x names in common, the array representation of the layered map will contain $m + n - x$ elements.

The first m elements of the layered array will be drawn from the elements defined by M's array. The remaining $n - x$ elements in the layered map will be drawn from a subset of the regions defined in N. This subset consists of those regions in N that are not bound to names also found in M.

The domain of the symbol-table for the map returned by the layer operation consists of the unique names defined by M and N; the indices to which these names are bound in the layered-map is computed as follows: names defined by M retain the same indices in the symbol-table built for the layered-map as they do in M's symbol-table; the indices of names defined by N that are in the layered-maps symbol-table is determined by adding the symbol's index in N's symbol-table to m, the starting offset in the layered-map's array where N's elements are added, and subtracting from this the number of regions in N before this region that are bound to names defined in M. The **count_indices** function computes this sum.

Note that the sharing rules found in Id are similar to those defined in Symmetric Lisp: scalar values defined by the two argument maps to **layer** are copied into the array defining the new layered map; non-scalar, structure values are shared.

Much of the code shown could actually be computed statically in many cases; for example, if the structure of **M** and **N** is known, it is possible to statically determine the intersection of their names as well as the structure of the layered map and the value of the indices to which names in the layered are to be mapped. Such optimizations would greatly simplify the translated version of the program.

7.7 Selection

The main selection operators **select** and **index** are easily translated into a suitable Id representation. The expression (**select M id**) would be represented as

```
{ (a,s) = < translation of M>;
  a1 = array(1,1);
  a1[1] = a[(get_val s id)]
in
   (a1,(make_symbol_table ((id,1))))}
```

The object returned by the above fragment is a tuple; the first element of the tuple is an array whose sole element contains the value of id in M and second element is a symbol-table that binds id to 1, the index in the array that contains its value.

The expression (**index M e**) is translated similarly:

```
{ (a,s) = < translation of M>;
  n = < translation of e>;
  a1 = array(1,1);
  a1[1] = a[n];
in
   (a1,(make_symbol_table ()))}
```

The **mlast** operator simply returns the value of the last element in the array used to hold the values of its argument map's regions. The expression (**mlast M**) is translated as:

```
{ (a,s) = < translation of M>;
   (1,j) = (bounds a)
  in a[j] }
```

7.8 Abstraction and Application

Kappa and lambda abstraction expressions in Symmetric Lisp have a straightforward representation in Id in terms of unnamed functions. The main difference between Symmetric Lisp's treatment of abstraction and Id's is the fact that Id functions are fully curried while Symmetric Lisp functions are not. It is responsiblity of the Symmetric Lisp to Id translator to check that the arity of a Symmetric Lisp function applied to n arguments is also n. With this assumption in mind, one can translate a Symmetric Lisp **lambda** expression of the form:

```
(lambda (x1 x2 ... xn)
   e1 e2 ... ek)
```

into an Id fragment of the form:

```
{fun x1' x2' ... xn' .
    < translation of
          (mlast
             (map x1 : x1'
                  x2 : x2'
                    ...
                  xn : xn'
                  e1 e2 ... ek))>}
```

where the **xi**′ are new variables.

A Symmetric Lisp function application, (**f a1 a2** ... **an**), is translated into the Id fragment:

```
(translation of f
    translation of a1
           ⋮
    translation of an)
```

A Symmetric Lisp application is simply an Id application in which the function and each of its arguments have been suitably translated. Note that this translation preserves the Symmetric Lisp sharing rule: if any of the **ai** evaluate to scalars, their *values* are simply bound to the corresponding formal in the translated version of **f**. Map structures, on the other hand, will get translated into *addresses*; thus, it will be possible for expressions in the body of the function to side-effect map objects since they will have access to their addresses.

For example, the function **fact** written in Symmetric Lisp:

```
fact : (lambda (n)
          (if (= n 0) 1
              (n * (fact (1- n)))))
```

would be translated into the following Id block (assume that "*" "-" and "=" are predefined and that the l-value of **fact** is (**f**,**i**)).

```
{fun n' .
  { (a,s) = { a = array (1,2);
              a[1] = n';
              a[2] = { (arr,index) = n;
                       in if arr[index] == 0 then 1
```

```
                              else arr[index] *
                                     (f[i] (arr[index] -1)) };
                    n = (a,1)
                  in (a, make_symbol_table (("n",1)) }
       (1,j) = bounds a
     in a[j] }}
```

The implementation of kappa abstraction is slightly different than the implementation of lambdas. A kappa abstraction of the form:

```
(kappa (x1 x2 ... xn)
   e1 e2 ... ek)
```

is translated into an Id fragment of the form:

```
{fun x1' x2' ... xn' .
    < translation of
          (map x1 : x1'
               x2 : x2'
                  ...
               xn : xn'
               e1 e2 ... ek))>}
```

where the xi′ are new variables; a **kappa** abstraction does not have an **mlast** as part of its body. The translation of kappa application follows the same rules as the translation of a lambda application.

7.9 Sequential Evaluation

The **seq** special form in Id can be used to implement the **seqmap** construct in Symmetric Lisp. A sequential map of the form:

```
(seqmap
   n1 : e1
   n2 : e2
    ⋮
   nk : ek)
```

can be translated into the following Id program fragment:

```
{ a = array(1,k);
  n1 = (a,1);
  n2 = (a,2);
    ...
  nk = (a,k)
  (seq
      a[1] = e1;
      a[2] = e2;
        ...
      a[k] = ek)
  in (a,(make_symbol_table (("n1",1) ("n2",2) ... ("nk",k))))}
```

The **seq** form prevents the evaluation of its i^{th} component expression until the $i-1^{st}$ expression has yielded a result. In the above example, the evaluation of `e2` will not begin until after `e1` has yielded a result and this result has become bound to `a[1]`. Note that all names defined by the **seqmap** are recorded, in the Id code, as part of the lexical environment in which the `ei` evaluate. This means that if `ei` requires the value of name `nj` in order to return a result where `j` $>$ `i`, a deadlock would arise; the evaluation of `ej` will never commence because `ei` will never return a result.

7.10 Filling an Empty Region

The **fill** operator is a generalization of the Id I-structure assignment operator. Unlike I-structure assignment, filling a region requires examining the region being filled to ensure that it is either empty or contains a value that is identical to the value that is to fill it. We introduce a new instruction called **empty?-and-set** that, given an I-structure element, i, and a value v, tests whether i is empty and if so, atomically sets i's contents to be v[2]; **empty?-and-set** returns true if the I-structure was empty and false otherwise.

The translation of the expression, `(fill x v)`, is shown below:

```
  { (arr,index) = x;
   in
    if (not empty?-and-set arr[index] v) then
       if arr[index] == v then v
          else  error }
```

(**error** is a special error symbol.)

There is a a similar translation for an expression of the form `(fill e1.x e2)`:

```
  { (a,s) = < translation of  e1>;
    v1 = (get_val s "x"); x1 = x
```

[2] The implementation of this operator is straightforward in a dataflow machine that allows the tag bits of a data structure to be extracted. One of the tag bits of interest in an I-structure is a *presence* bit that indicates whether the I-structure has a value or not. The **empty?-and-set** predicate can be compiled into a primitive dataflow machine instruction that extracts the presence bit field of its argument and performs an assignment on the I-structure if needed.

```
in
  { (arr,index) = (if v1 =  undef then x1 else v1);
   in if (not empty?-and-set arr[index] v) then
      (if arr[index] == v then v
          else  error)}}
```

There is a similar translation for index selection.

The translation for assignment expressions is similar to that given above for `fill` with two slight changes: (1) There is no equality test on the element being set. The translation for `set` does not check whether is component structure is empty or not. (2) Since assignment can change the contents of a cell which already has a value, we need a new assignment operator (written `:=`) that replaces the contents of an array cell without checking whether the cell already has been assigned a value.

7.11 Implementing Locks

The `holding` expression when used in conjunction with locks can be used to express applications where mutual exclusion of a shared resource is required. We can implement locks in a dataflow system via I-structures. The basic observation in the implementation of locks and mutual exclusion is that an I-structure whose presence bit is not set prevents an expression from accessing the I-structure's contents; thus, by manipulating presence bits in various way, we can use I-structures as semaphore-like objects.

To manipulate locks, we introduce two new primitive instructions. The first, `set-presence-bit`, takes as its argument a tuple, `(l,i)`, that represents the l-value of an I-structure element and sets the presence bit of that element "on" regardless of whether there is an element in the I-structure or not.

The second instruction, `read-lock`, given an l-value of an I-structure, `(l,i)`, tries to read the I-structures contents; if the presence bit of the I-structure is "on", *i.e.*, if the structure is readable, the instruction sets the presence bit "off" thereby making the I-structure element unreadable. The instruction itself returns no values; it simply sends signals to its destinations when it executes.

For example, consider the following program fragment:

```
 x : lock
```

```
(holding x (set y (1+ y)))
```

that acts like an atomic counter. The binding, `x : lock`, is translated as follows (assume that `x`'s address is `(l,i)`):

```
l[i] = array(1,1);
(set-presence-bit (l,i))
```

A lock is simply a one-element array; the `set-presence-bit` instruction makes the lock accessible to other expressions.

The `holding` expression is now translated thus:

```
(seq
   (read-lock (l,i))
   { (arr,index) = y;
        in arr[index] := arr[index] + 1};
   (set-presence-bit (l,i)))};
```

The `holding` expression is translated into a `seq` form. The `seq` construct evaluates its component expressions in order. In the abstract, the `read-lock` operator *atomically* reads the I-structure and sets the structure's presence bit off; this effectively preventing any other expression from reading the structure. In the abstract, the `set-presence-bit` instruction resets the lock enabling other `read-lock` instructions to access it.

The above scenario is complicated by the fact that there may be many `read-lock` instructions simultaneously evaluating. We must, therefore, address the question of what happens if a `read-lock` instruction finds that the I-structure is not readable – such a situation would arise if another `read-lock` instruction executed prior to this one. In this case, the instruction is placed on a waiting queue. There may be potentially many such instructions that attempt to read the lock; each instruction is placed on the same wait queue.

We must also take care to prevent race conditions once the lock is released. To avoid a race condition, we ensure that, before the presence bit of the lock is turned back on, there are no `read-lock` instructions waiting on the queue associated with this I-structure. If there are waiting instructions, we simply remove the first one, and send a signal to its destinations; the presence bit of the lock need never be turned on in this case since there are instructions waiting to simply turn it back off. Thus, the implementation of the `set-presence-bit` instruction must check that there are no instructions waiting on the lock before actually setting the presence bit. Note that this solution prevents starvation (since waiting `read-lock` instructions are removed in FIFO order) as well as race conditions (since the presence bit is set on only if there are no instructions waiting to access the I-structure element).

The translation of a holding expression of the form, `(holding` l e`)`, where e is a map expression or an expression that contains a map (*e.g.*, a lambda application) is much more complicated. To guarantee serializability, we require that every sub-expression of e terminate before the lock is released. A *termination signal* of some sort must be received from each sub-expression of e before the **reset** operation can be executed. This signal indicates that a *value* has been produced by the sub-expression. The **terminate?** operator, given an object x, blocks if x is not a value and returns true otherwise. If x is a scalar, it returns true immediately; if x is a map object, it returns true when each of the map's component elements are values. Thus, if a map M has as a sub-expression another map N, then, by definition, **(terminate? M)** returns true only when all of M's sub-expressions and all of N's sub-expressions yield values. This constraint on parallelism is very severe and it remains to be seen whether it is viable on a mulitprocessor dataflow machine[3].

7.12 Summary

The translation sketch given in this chapter exploits the lexical scoping rules found in Id to model as much of Symmetric Lisp's own name lookup-rule as possible. There are two places, however, where a standard lexical name-lookup rule is not sufficient or appropriate. The first is in the translation of scope-expressions. Since the lexical-evaluation environment of an expression can be superseded by embedding the expression inside a with form, some mechanism that records the names defined by a map expression is necessary. The symbol-table abstraction was used for this purpose. A symbol-table effectively bridges program-structure names with data-structure ones; every identifier in a Symmetric Lisp program has two representations in the Id translation: the first as a program identifier found in the block representing the map expression in which it is defined and the second as a string in the symbol table that serves as the representation of the map data-structure.

The other area where the Id scope-rules and naming conventions are not appropriate is in the translation of expressions that dynamically create new names. The **read** special-form is

[3]We note that implementing such an operation, while not easily expressible in a high-level language like Id is surely implementable in the base language of dataflow graphs.

characteristic of this class of expressions. To make the translation truly robust, we must ensure that an erroneous Symmetric Lisp program does not result in Id-based runtime errors. In particular, a **read** expression that returns a map object that refers to a free name n not declared anywhere in the translation will cause an Id runtime error indicating that the Id variable n is unknown. A more desirable solution would be to have this error captured by the virtual Symmetric Lisp machine that runs on top of Id; it is this same machine that interprets error symbols, handles exceptions etc. To support the translation of dynamically-generated names we must require that the value of every name be mediated by a lookup function; in this case, we must sacrifice the simplicity afforded by exploiting Id's static scoping discpline in implementing Symmetric Lisp's lexical name lookup rule in favour a more generalized lookup rule. Instead of translating maps into Id blocks, maps must be translated into an environment abstraction structure; names would need to be translated into strings and each name reference regardless of whether found in a scope-expression or in a simple map expression must be translated via the lookup function. The structure of this translation would look very similar to the structure of the abstract interpreter that defines the operational semantics of the language.

In terms of giving insight into the parallel evaluation semantics of Symmetric Lisp, our translation sketch can be used to understand how Symmetric Lisp evaluation semantics compares to other fine-grained parallel languages. The translation omits, however, many important low-level details. We have ignored issues relating to code-mapping, scheduling policies among concurrent processes, and resource management. We feel, however, that these issues are endemic to parallel systems in general and, consequently, we expect that solutions to these problems developed for, say a dataflow system intended to implement Id, can be used to good effect in the implementation of Symmetric Lisp as well. Devising solutions to these questions forms an important area of future research.

Chapter 8

Conclusions

We began our investigation of symmetric languages on the premise that program structures and data structures are related to one another in a strong sense. We posited that the fundamental relationship between these structures lie in their implicit support for the construction and manipulation of *naming environments.* Given this observation, we posed the question, "What is the structure of a programming language that is based on a model in which there is *no* distinction between program structures and data structures?" Since our hypothesis regarding the similarity of program structures to data structures rested on the fact that both define naming environments of one sort or another, we proposed a programming model (called the *symmetric* model) whose fundamental program structure has the same semantics and representation as its fundamental data structure. In the symmetric model, program structures and data structures behave as *first-class naming environments.*

We chose to examine the behaviour of a symmetric language in the context of a non-strict evaluation model: the component elements of a program or data structure may be examined even as other elements in the structure continue to evaluate. Naming environments are, thus, parallel program and data structures.

8.1 Contributions

The main thrust of thesis has been in the area of programming language design. Language design, by its very definition, is a very subjective endeavour; design decisions are often governed as much by subjective biases on the part of the designer as they are by objective scientific criteria. Nonetheless, there are some important objective metrics that are generally accepted by which language designs have traditionally been judged.

One important metric is *expressivity*; in our view, an expressive language is simple (*i.e.*, it contains relatively few primitives) *and* its "phrasebook" is short - little verbiage is required to define within the language all the programming phrases a programmer is likely to want. Expressivity has traditionally been at odds with *simplicity* in the sense that simple languages often induce bulky or complex programs outside fairly narrow domains.

Much contemporary work in programming language design has focussed on trying to reconcile these two goals and there have been several highly successful results: Pascal and Scheme are two good examples.

While our goal in investigating the design of symmetric languages has been in the spirit of these language design efforts, our approach and specific interests have been much different. Unlike either Pascal or Scheme which were designed partially as a reaction against their complicated (albeit expressive) contemporaries (PL/1 and Algol-68 for Pascal and MacLisp for Scheme), the design of Symmetric Lisp was not intended to offer an alternative to any one language; it is the outgrowth of the initial observation that the separation of program-structures and data-structures in modern languages (in general) hinders expressivity and introduces complexity.

The idea of eliminating complexity by designing a uniform and self-consistent language is by itself not a unique thought; functional languages eliminate complexity by treating functions as their basic program object; object-oriented languages eliminate data structures by transforming all objects into programs. To our knowledge, however, the idea of achieving uniformity by representing program structures as data objects has not been studied. We, therefore, expect that language designers may find the programming model developed in this thesis to be a useful thought-tool in which to understand the relationship between program structures and data structures found in many diverse languages. Because we have chosen to consider symmetric languages in the presence of a non-strict evaluation model, we also expect that the results described in this thesis can be used to understand competing parallel programming paradigms as well.

8.1.1 Summary

Broadly speaking, then, the main contribution of this thesis has been to give a detailed account of the semantics, implementation and application of a language based on the symmetric model. The thesis has argued that such a model provides a unified framework upon which to understand the relationship between many superficially-different program and data structures found in modern languages. Moreover, the thesis also contends that such a model supports certain novel programming paradigms not easily represented within other existing models. Specifically, the thesis has addressed the following questions:

1. What is the structure of a symmetric language program and what are the operators that should form the kernel of a symmetric language?

2. What is the formal semantics of a symmetric language? What does the formal semantics tell us about how symmetric languages are related to languages based on more classical models?

3. What are the kinds of uniformity that the symmetric model supports and what is the impact of such uniformity on program methodology and construction?

4. What is the ramification of first-class naming environments on type-inference and name-translation?

5. What constraints does a symmetric language's non-strict evaluation semantics impose on the semantics of the base language to which it is compiled? What linguistic mechnaisms must a base language provide to efficiently support a symmetric language?

Chapter 2 and 3 addressed the first question by presenting the semantics of Symmetric Lisp. We defined the **map** as the fundamental environment constructor and showed how maps can be used to serve as both program structures as well as data structures.

Chapter 4 examined two manifestations of program/data uniformity: (1) uniformity of different program and data constructs and (2) uniformity of data and process. We examined how superficially different program structures found in other languages can be translated into Symmetric Lisp. We showed how blocks, packages, inheritance systems, guarded commands, and lazy data structures could be all thought of as structures that manipulate naming environments; each of these structures were translated into operations over Symmetric Lisp maps. As an example of programming paradigms ideally suited to the symmetric model we developed a new program structure called the parallel process lattice. Externally, the lattice was treated as a large data structure whose fields represented data points of one sort or another; internally, it consisted of a collection of long-lived concurrent processes. Both the lattice-as-a-process view as well as the lattice-as-a-structure view are accomodated equally well by implementing the lattice as a map. This chapter also examined the role of maps as a parallel process encapsulator; among other things, we showed how maps could be used to implement cyclic networks and resource managers.

Chapter 5 examined yet another manifestation of uniformity: the uniformity of the command-level domain and application-level domain. We sketched the design of a concurrent interpretative programming environment in which Symmetric Lisp acts as the base language. Using first-class parallel environments we showed how one could build a concurrent Symmetric Lisp front-end. First-class environments can be structured to serve as file-systems. Non-strict evaluation allows one to build daemon processes that can monitor different states of the programming environment non-intrusively.

The last two chapters of the thesis addressed questions of compilation and implementation. In Chapter 6, we developed an extended polymorphic type-inference system for Symmetric Lisp called an *environment-inferencing system* that could be used as the basis for implementing a procedure to statically determine the proper evaluation environment of identifiers in a restricted subset of the language.

The non-strict evaluation semantics of maps makes the language of dataflow graphs an ideal candidate to serve as the base language for a Symmetric Lisp implementation. Chapter 7, therefore, presented a translation scheme from Symmetric Lisp into Id[53] a high-level parallel language whose compilation into dynamic dataflow graphs has been well-studied.

8.2 Directions for Future Research

Although this thesis has given a comphrehensive description of Symmetric Lisp, there are a number of issues relating to language design, programming environments, compilation and implementation of symmetric languages in general that have yet to be addressed. We examine some of these issues below.

Language Design

There are a few important language-related issues that have not been addressed fully in the thesis.

The symmetric model was presented in terms of maps and operations over them. The particular topology chosen for maps was a simple linear array of regions. Neither the representation of programs as maps nor the topology of maps as vectors are fundamental to the symmetric model *per se*; one could conceivably have chosen alternative representations and topologies that also provide the necessary foundation for modelling program and data objects. Alternative representations for program structures within the framework of the symmetric model is an interesting avenue of research.

There are a number of open issues dealing directly with Symmetric Lisp. The first issue has to do with the implementation of an exception handling mechanism. We saw an example where exceptions would be useful on page 138 wherein a file-lookup operation implemented via `with` should raise an error if the file in question is not found in the specified directory; many other examples can be easily brought to mind. Exceptions are a restricted form of general continuations[32] and it is yet to be seen whether there exists a reasonable continuation-based semantics for parallel non-strict languages. Non-strictness and continuations interact in the following sense: a non-strict expression spawns concurrent tasks for each of its sub-expressions; if a sub-expression raises an exception to handle an error, however, it is often desirable to be able to stop other concurrently evaluating sub-expressions. The mechanism by which this is to be done is a topic for investigation.

Secondly, we proposed locks as a primitive data object to be used when access to a shared mutuable structure must be serialized. The strict semantics of a holding expression, however, can lead to severe constraints on concurrency. It is not yet clear that this is a viable mechanism in the context of what is otherwise a fine-grained, non-strict evaluation model. The implementation costs of locks and `holding` expressions have yet to be assessed.

As a related issue, we lack experimental data on the cost of using the open-map abstraction as the basic stream communication mechanism. The overhead involved in implementing open-maps as stream abstractions in which each stream element is a map and in which each stream access must be mediated via a lock is yet to be determined.

The Programming Environment

We argued in Chapter 5 that first-class environments can be used to model file-systems in a Symmetric Lisp language-based workstation. There are a number of important properties of file-systems, however, that Symmetric Lisp environments do not possess. Of these, the most important are (1) support for persistence and (2) support for protection.

Support for persistency requires mechanism to guarantee recovery of files in the event of system crashes; support for protection requires mechanisms to ensure that protected files can be only accessed as specified by their creator. Symmetric Lisp does not adequately address the persistency question because we have not yet a developed a data recovery scheme for Symmetric Lisp maps. Maps by themselves do not pose any special problems in the implementation of persistence, but the fact that they are the only structure object in the language raises some important questions on garbage collection of small objects found on a persistent store.

Secondly, the protection facilities provided by the **priv** prefix is undermined by the fact that the contents of regions can be selected by position as well as by name. The model of protection in Symmetric Lisp is based on user-agreed convention; there is no means to enforce a protection policy given the semantics of maps as they are currently defined.

Compilation

There are several important compilation issues that we did not address in the thesis. The first has to do with augmenting the type-inference system to support side-effects. We hinted at a possible approach to supporting side-effects that works by keeping track of which objects are mutable and which are not, but there are many subtle details in this proposal that have to be addressed.

We also omitted the proof of the soundness theorem given in Chapter 6 as well as the construction of the algorithm to compute principal types. Extending the type system to support interpreter-based operations such as **read** is also an important topic for future research.

Implementation

Finally, there are many open questions still remaining on the system-level implementation of symmetric languages. These questions concern architectural support for environment structures, process scheduling and communication, and resource management policies for environments. We presented one high-level implementation approach in terms of dataflow systems; there are still many unanswered questions about the implementation of Symmetric Lisp on alternative architectures.

In Closing

We forsee future work on symmetric languages concentrating primarily on the issues cited above. This is not to suggest that there is no further work remaining on the semantics of symmetric languages or in understanding the paradigms they support. Although the ultimate success of the symmetric language effort will be measured in terms of how sucessful we are in devising satisfactory solutions to these open questions, we believe that, independent of any practical realization, symmetric languages offer themselves as an expressive thought-tool that present programming languages in a new and interesting light.

Bibliography

[1] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] William Ackerman and Jack Dennis. VAL – A Value-Oriented Algorithmic Language: Preliminary Reference Manual. Technical Report 218, MIT, 1979.

[3] **Reference Manual for the ADA Programming Language**, 1982.

[4] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* PhD thesis, MIT Artificial Intelligence Laboratory, 1985. Published as AI-TR-844.

[5] Arvind and J.D. Brock. Resource managers in functional programming. *Journal of Parallel and Distributed Computing*, 1(1):5–21, 1984.

[6] Arvind and David Culler. *Dataflow Architectures*, volume 1, pages 225–253. Annual Reviews Inc., 1986.

[7] Arvind, Rishiyur Nikhil, and Keshav Pingali. I-Structures: Data Structures for Parallel Computing. In *Proceedings of the Workshop on Graph Reduction.* Springer-Verlag, 1986. Lecture Notes in Computer Science, Number 279.

[8] Arvind and Rishiyur S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. In *Proceedings of the PARLE Conf.* Springer-Verlag, 1987. Lecture Notes in Computer Science, LNCS 259.

[9] E.A. Ashcroft and W.W. Wadge. Lucid, a Nonprocedural Language with Iteration. *Communications of the ACM*, 20(7):519–526, July 1977.

[10] M. Atkinson and R. Morrison. Types, Bindings, and Parameters in a Persistent Environment. In *Persistence and Data Types Papers for the Appin Workshop.* University of St. Andrews, August 1985.

[11] Malcolm Atkinson and R. Morrison. Procedures as Persistent Objects. *ACM Transactions on Programming Languages and Systems*, 7(4):539–559, October 1985.

[12] John Backus. Can Programming be Liberated from the Von Neumann Style: A Functional Style and Its Algebra of Programs. *Communications of the ACM*, 21(8):613–640, 1978.

[13] H. Barendregt. *The Lambda Calculus.* North-Holland, 1981.

[14] Daniel Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops:Merging Lisp and Object-Oriented Programming. In *Object Oriented Programming Systems, Languages and Applications*, pages 17–30, September 1986.

[15] Robert Burstall and Butler Lampson. A Kernel Language for Modules and Abstract Data Types. In *International Symposium on Semantics of Data Types*. Springer-Verlag, 1984. Lecture Notes in Computer Science, Number 173.

[16] Luca Cardelli. A Semantics of Multiple Inheritance. In *International Symposium on Semantics of Data Types*. Springer-Verlag, 1984. Lecture Notes in Computer Science, Number 173.

[17] Luca Cardelli. Amber. Technical Report 11271-840924-10TM, AT&T Bell Laboratories, 1984.

[18] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[19] Nick Carriero and David Gelernter. Application Experience with Linda. In *First Annual Proceedings of the ACM Symposium on Parallel Programming*, July 1988.

[20] Nick Carriero, David Gelernter, and Jerry Leichter. Distributed Data Structures in Linda. In *13th ACM Symposium on Principles of Programming Languages*, Jan. 1986.

[21] Chris Clack and Simon Peyton-Jones. Strictness Analysis: A Practical Approach. In *1985 Proceedings on Functional Programming Languages and Computer Architecture*, pages 35–49. Springer-Verlag, 1985. Lecture Notes in Computer Science, Number 201.

[22] E.F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[23] O.J. Dahl, B. Myhruhaug, and K. Nygaard. The Simula67 Base Common Base Language. Technical report, Norwegien Computing Center, 1970.

[24] Luis Damas and Robin Milner. Principle Type-schemes for Functional Programs. In *9th ACM Symposium on Principles of Programming Languages*, pages 207–212, January 1982.

[25] Jack Dennis. An Operational Semantics for a Language with Early Completion Structures. In *Conf. on Formalization of Programming Concepts*, April 1981.

[26] Jack Dennis, Joseph Stoy, and Bhaskar Guharoy. VIM: An Experimental Multi-User Computer System Supporting Functional Programming. In *1984 Conf. on High-Level Computer Architecture*, 1984.

[27] Peter Deutsch and Allan Schiffman. Efficient Implementation of the Smalltalk-80 System. In 10th *ACM Symposium on Principles of Programming Languages*, pages 297–302, 1983.

[28] Edsger Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[29] Lee Erman, Fredrick Hayes-Roth, Victor Lesser, and D. Raj Reddy. The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty. *Computing Surveys*, 12(2), 1980.

[30] K.P. Eswaren, J.N Gray, R.A. Lorie, and I.L Traiger. The Notion of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, 1976.

[31] William Clinger *et. al.* The Revised Revised Revised Report on Scheme or An UnCommon Lisp. Technical Report AI-TM 848, MIT Artificial Intelligence Laboratory, 1985.

[32] Daniel Friedman, Chris Haynes, and Eugene Kohlbecker. Programming With Continuations. In *Program Transformations and Programming Environments.* Springer-Verlag, 1985.

[33] Daniel P. Friedman and David Wise. Cons Should Not Evaluate Its Arguments. In *Automata, Languages and Programming*, pages 257–284. Edinburgh University Press, 1976.

[34] R. Gabriel and J. McCarthy. Queue-Based Multi-Processing Lisp. In *Proceedings of the 1984 Conf. on Lisp and Functional Programming*, pages 25–44, August 1984.

[35] David Gelernter, Nick Carriero, Sarat Chandran, and Silvia Chang. Parallel programming in Linda. In *International Conference on Parallel Processing*, pages 255–263, August 1985.

[36] G. Gentzen. Investigations into Logical Deduction. In M.E.Szabo, editor, *The Collected Papers of Gerhard Gentzen.* North-Holland Press, 1969.

[37] Dave Gifford and John Lucassen. Integrating Functional and Imperative Programming. In *Proceedings of the 1986 Conf. On Lisp and Functional Programming*, pages 28–39, 1986.

[38] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Press, 1983.

[39] R. Greenblatt, T. Knight, J. Holloway, D. Moon, and D. Weinreb. The LISP Machine. In *Interactive Programming Environments*, pages 326–352. McGraw-Hill, 1984.

[40] Robert Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[41] J. Heering and P. Klint. Towards Monolingual Programming Environments. *ACM Transaction on Programming Languages*, 7(3):183–213, July 1985.

[42] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[43] Paul Hudak and David Kranz. A Combinator-Based Compiler for a Functional Language. In 11^{th} *ACM Symposium on Principles of Programming Languages*, pages 122–132, January 1984.

[44] J.C.Shaw. JOSS:A Designer's View of an Experimental On-Line Computer System. In *AFIPS, 1964 Fall Joint Computer Conference*, pages 455–464, 1964.

[45] J.W.Klop. Term Rewriting Systems: A Tutorial. *Bulletin of the European Association of Theoretical Computer Science*, 1987.

[46] Guy L. Steele Jr. and W. Daniel Hillis. Connection Machine Lisp: Fine-Grained Parallel Symbolic Computing. In *Proceedings of the 1986 Conf. on Lisp and Functional Programming*, pages 279–298, August 1986.

[47] Barbara Liskov and Robert Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[48] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.

[49] Albert Meyer and Mark Reinhold. "Type" is not a Type: A Preliminary Report. In 13th *ACM Symposium on Principles of Programming Languages*, pages 287–296, January 1986.

[50] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, December 1978.

[51] Robin Milner. The Standard ML Core Language. Technical Report CSR-157-84, Edinburgh University, 1984.

[52] P. Naur and M. Woodger (Eds.). Revised Report on the Algorithmic Language Algol 60. *Communications of the ACM*, pages 1–20, June 1963.

[53] Rishiyur Nikhil. ID Reference Manual (Version 88.0). Technical report, MIT, 1988. Computation Structures Group Technical Report.

[54] Kent Pitman. The Revised MacLisp Manual. Technical Report 295, MIT, 1983.

[55] Ehud Shapiro. Concurrent Prolog: A Progress Report. *IEEE Computer*, 19(8):44–60, August 1986.

[56] Brian Smith and J. des Rivières. The Implementation of Procedurally Reflective Languages. In *Proceedings of the 1984 Conf. on Lisp and Functional Programming*, pages 331–347, August 1984.

[57] Guy Steele Jr. *Common Lisp: The Language*. Digital Press, 1984.

[58] Guy Steele Jr. and Gerry Sussman. Lambda: The Ultimate Imperative. Technical Report AI-TM 353, MIT Artificial Intelligence Laboratory, 1976.

[59] Guy Steele Jr. and Gerry Sussman. The Art of the Interpreter, or the Modularity Complex. Technical Report AI-TM 453, MIT Artificial Intelligence Laboratory, 1978.

[60] Norihisa Suzuki. Inferring Types in Smalltalk. In 8th *ACM Symposium Principles of Programming Languages Conf.*, pages 187–199, January 1981.

[61] Daniel Swinehart, Polle Zellweger, Richard Beach, and Rober Hagmann. A Structural View of the Cedar Programming Environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–491, October 1986.

[62] W. Teitelman and L. Masinter. The Interlisp Programming Environment. *Computer*, 14(4):25–34, April 1981.

[63] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, Department of Computer Science, 1988. Published as CST-52-88.

[64] Kenneth Traub. A Compiler for the Tagged-Token Dataflow Architecture. Technical Report TR-370, Massachusetts Institute of Technology, Laboratory for Computer Science, 1986.

[65] D. Turner. Miranda: A Non-Strict Functional Language with Polymorphic Types. In *1985 Proceedings on Functional Programming Languages and Computer Architecture*, pages 1–16. Springer-Verlag, September 1985. Lecture Notes on Computer Science, Number 201.

[66] D. A. Turner. A New Implementation Technique for Applicative Languages. *Software - Practice and Experience*, 9:31–49, 1979.

[67] Mitchell Wand and Daniel Friedman. The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower. In *Proceedings of the 1986 Conf. on Lisp and Functional Programming*, pages 298–307, August 1986.

[68] David H. Warren. Logic Programming and Compiler Writing. *Software Practice and Experience*, 10(2):97–127, February 1980.

[69] K.-S. Weng. An Abstract Implementation for a Generalized Data Flow Language. Technical Report TR-228, Laboratory for Computer Science, MIT, Cambridge, Mass., 1979.

[70] C. Wetherell. Error Data Values in the Data-Flow Language VAL. *ACM Transactions on Programming Languages and Systems*, 4(2):226–238, 1982.

[71] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, 1985.

OFFICIAL DISTRIBUTION LIST

Director Information Processing Techniques Office Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	2 copies
Office of Naval Research 800 North Quincy Street Arlington, VA 22217 Attn: Dr. R. Grafton, Code 433	2 copies
Director, Code 2627 Naval Research Laboratory Washington, DC 20375	6 copies
Defense Technical Information Center Cameron Station Alexandria, VA 22314	12 copies
National Science Foundation Office of Computing Activities 1800 G. Street, N.W. Washington, DC 20550 Attn: Program Director	2 copies
Dr. E.B. Royce, Code 38 Head, Research Department Naval Weapons Center China Lake, CA 93555	1 copy